



Universidad
Carlos III de Madrid

Departamento de Telemática

PROYECTO FIN DE CARRERA

DESARROLLO DE APLICACIONES PARA WINDOWS PHONE

Autor: Francisco José Castellanos de la Torre

Tutor: Estrella María García Lozano

Leganés, julio de 2013

Título: Desarrollo de aplicaciones para Windows Phone

Autor: Francisco José Castellanos de la Torre

Director: Estrella María García Lozano

EL TRIBUNAL

Presidente: M^a Celeste Campo Vázquez

Vocal: Rubén Cuevas Rumín

Secretario: José Arturo Mora Soto

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 5 de Julio de 2013 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO

PRESIDENTE

Resumen

Desarrollar una aplicación para dispositivos móviles no es una tarea sencilla, ya que precisamos no solo tener conocimientos de programación, sino que también tenemos que familiarizarnos muy bien con el entorno de desarrollo y el sistema operativo para el cual pretendemos crear la aplicación. Esto consiste en conocer en profundidad tanto las características técnicas de los dispositivos como las distintas APIs que tenemos a nuestra disposición.

Hemos elegido Windows Phone 7.5 porque ha sido el último sistema operativo en llegar al mercado y ofrece interesantes novedades respecto a los otros dos grandes competidores, como son iOS, de Apple y Android, de Google. Por tanto, todavía es un gran desconocido y no existen muchos artículos que expliquen el desarrollo de aplicaciones.

La explicación del desarrollo de la aplicación se centrará en el apartado lógico, ya que ésta es la gran cualidad de este sistema operativo. La de ofrecer un buen contenido antes que un buen diseño. Esto no quiere decir que dejemos de lado el diseño, sino que éste será sencillo.

En este proyecto, se pretende explicar cómo desarrollar una aplicación móvil desde cero para el sistema operativo Windows Phone 7.5 y su posterior publicación en el Marketplace, de manera que quede disponible para todo el mundo.

Palabras clave: aplicaciones para móviles “apps”; Windows Phone 7.5; desarrollo; publicación en el mercado de aplicaciones.

Abstract

Developing a mobile application is not an easy task. We need programming skills and we have to familiarize well with the development environment and operating system for which we want to create the application. This is, to know in depth the technical characteristics of the devices and the different APIs we have at our disposal.

We chose Windows Phone 7.5 because it was the last operating system to hit the market and offers interesting new features compared to the other two big competitors such as iOS (Apple) and Android (Google). So it is still a great unknown and there are no many articles that explain the development of applications.

The explanation of the development of the application will focus mainly in the logical section, as this is the best feature of this operating system. This feature is to provide good content rather than good design. This does not mean we shake off the design, but this will be easy.

This project aims to explain how to develop a mobile application for the Windows Phone 7.5 operating system and its subsequent publication in the Marketplace, so that it is available for everyone.

Keywords: mobile applications “apps”; Windows Phone 7.5; development; publication in the application market.

Índice de Contenido

1. INTRODUCCIÓN	2
I. Introducción.....	2
II. Objetivos.....	2
III. Fases del desarrollo	3
IV. Medios empleados.....	4
V. Estructura de la memoria	4
2. ESTADO DEL ARTE.....	5
I. Android	5
II. iOS.....	5
III. Blackberry OS.....	6
IV. Windows Phone	6
V. Firefox OS y Ubuntu Phone OS	7
3. INTRODUCCIÓN AL DESARROLLO DE APLICACIONES PARA WINDOWS PHONE	8
I. Arquitectura de Windows Phone	8
II. Ciclo de vida de una aplicación.....	11
III. Actualización Windows Phone 7.8.....	12
IV. Lenguaje de desarrollo	13
a) XAML: El lenguaje de la presentación	13
b) C#: El lenguaje de la lógica de negocio.....	14
V. Instalación del entorno de trabajo	15
a) Requisitos del sistema	15
b) Instalación del SDK.....	15
c) Instalación de paquetes extras	16
4. DESARROLLO DETALLADO DE UNA APLICACIÓN COMPLEJA CON VARIEDAD DE FUNCIONALIDADES	18
I. Empezar a desarrollar una aplicación.....	18
.....	23
a) Emulador de Windows Phone	23
b) Application Deployment Tool	25
c) Developer Registration Tool	26

II.	Patrón MVVM	27
III.	Estilo y diseño de las páginas.....	30
	a) Elementos	30
	b) Controles.....	30
	c) Recursos.....	32
	d) Estilos	34
	e) Plantillas.....	34
IV.	Página principal de nuestra aplicación: navegación, comandos	36
	a) Interfaz: MainPage.xaml	36
	b) Comandos	37
	c) Vista Modelo de la página: VMMainPage.cs	38
V.	Navegación en MVVM	39
VI.	Navegación hacia atrás	40
VII.	La base de datos	41
VIII.	Página donde gestionar las planificaciones de viaje: Barra de aplicación, ListBox, enlace a datos	47
	a) Página Planificacion.xaml	47
	b) Barra de aplicación	49
	c) Métodos de navegación en <i>code behind</i>	51
	d) Vista modelo de la página Planificacion.xaml	52
	e) La propiedad <i>Mode</i>	54
	f) Sistema de notificación de cambios.	55
IX.	Pantalla para crear una nueva planificación: DatePicker, inserciones en la BD, <i>tombstoning</i>	56
	a) Página NuevoPlan.xaml	56
	b) Herramienta para selección de fecha: <i>DatePicker</i>	57
	c) Vista modelo de la página NuevoPlan.xaml: interacción con la BD	61
	d) Método para calcular periodos de tiempo	64
	e) Teclado numérico	64
	f) <i>Tombstoning</i>	65
X.	Pantalla de edición de una planificación: <i>Pivot</i> , mapas.....	69
	a) EditaPlan.xaml: Primera vista del <i>Pivot</i>	69
	b) EditaPlan.xaml: Segunda vista del <i>Pivot</i>	72
	c) EditaPlan.xaml: Tercera vista del <i>Pivot</i> , mapas	72

d)	Vista modelo de la página EditaPlan.xaml: servicios web de Microsoft, geolocalización, recuperar datos de la BD	74
e)	Detalle de los días.....	85
XI.	Modo “Durante el viaje”	86
XII.	Página con las utilidades durante el viaje: cámara, ListPicker, múltiples ApplicationBar, ListBox, ExpanderView, brújula	87
a)	Interfaz de la vista Gastos.....	88
b)	Construcción de los elementos visuales de la vista gastos.....	89
c)	Vista modelo de la primera vista del Pivot: Gastos	92
d)	Interfaz de la vista Lista de gastos: ListBox.....	99
a)	Construcción de los elementos visuales de la vista lista de gastos.	99
b)	Construcción de los elementos visuales de la vista lista de gastos: ExpanderView	101
c)	Brújula.....	105
d)	Vista modelo de la tercera vista del Pivot: navegación.....	108
5.	PUBLICACIÓN DE UNA APLICACIÓN EN EL MARKETPLACE.....	111
I.	Crear una cuenta de Marketplace	111
a)	Tipo de cuenta	112
b)	Detalles de la cuenta	112
c)	Opciones de pago	112
II.	Probando la aplicación.....	112
a)	Detalles de la aplicación	113
b)	Pruebas automáticas	114
c)	Pruebas supervisadas	114
d)	Pruebas manuales.....	115
III.	Publicando la aplicación	116
6.	CONCLUSIONES.....	119
7.	PRESUPUESTO.....	120
8.	REFERENCIAS	121
a)	Sitios web.....	121
b)	Libros consultados	122

Índice de ilustraciones

ILUSTRACIÓN 1. ESPECIFICACIONES CHASSIS I.....	9
ILUSTRACIÓN 2. MODELO DE SOFTWARE EN WINDOWS PHONE.....	9
ILUSTRACIÓN 3. ESQUEMA DEL <i>RUNTIME</i> DE APLICACIONES DE WINDOWS PHONE.....	10
ILUSTRACIÓN 4. DIAGRAMA DE VIDA DE UNA APLICACIÓN EN WINDOWS PHONE.....	11
ILUSTRACIÓN 5. NUEVA GAMA DE COLORES DE ÉNFASIS.....	13
ILUSTRACIÓN 6. CABECERA DE LA PÁGINA DE DESCARGA DE NUGET.....	16
ILUSTRACIÓN 7. CONSOLA DE ADMINISTRACIÓN DE PAQUETES.....	17
ILUSTRACIÓN 8. EJEMPLO DE ÁREA DE TRABAJO DE VISUAL STUDIO 2010.....	18
ILUSTRACIÓN 9. ARCHIVOS DE UN PROYECTO.	19
ILUSTRACIÓN 10. COMPOSICIÓN DEL MARCO EN WINDOWS PHONE.....	20
ILUSTRACIÓN 11. ASPECTO FINAL DE LA APLICACIÓN.....	23
ILUSTRACIÓN 12. SELECCIÓN DEL DESTINO DE EJECUCIÓN.	23
ILUSTRACIÓN 13. ACCESO A LAS HERRAMIENTAS ADICIONALES DEL EMULADOR.	24
ILUSTRACIÓN 14. EMULADOR DE GPS ENVIANDO POSICIONES A NUESTRA APLICACIÓN.....	24
ILUSTRACIÓN 15. ENVÍO DE DATOS DE ACELERÓMETRO A NUESTRO EMULADOR.	25
ILUSTRACIÓN 16. HERRAMIENTA DE DESPLIEGUE DE APLICACIONES.....	26
ILUSTRACIÓN 17. HERRAMIENTA DE REGISTRO DE DISPOSITIVOS.	27
ILUSTRACIÓN 18. ESQUEMA MVVM.	28
ILUSTRACIÓN 19. MAINPAGE.XAML.....	36
ILUSTRACIÓN 20. AGREGAR REFERENCIA AL ENSAMBLADO.....	43
ILUSTRACIÓN 21. AGREGAR REFERENCIA AL ENSAMBLADO.....	44
ILUSTRACIÓN 22. MAINPAGE.XAML Y PLANIFICACION.XAML.....	47
ILUSTRACIÓN 23. PLANIFICACION.XAML.....	48
ILUSTRACIÓN 24. BOTONES EDITAR Y ELIMINAR HABILITADOS.....	48
ILUSTRACIÓN 25. BOTONES Y MENÚS DEL ELEMENTO <i>APPLICATIONBAR</i>	49
ILUSTRACIÓN 26. PROPIEDADES DEL ELEMENTO <i>APPLICATIONBARICONBUTTON</i>	50
ILUSTRACIÓN 27. PÁGINA CON LOS DISTINTOS DESTINOS YA CREADOS.....	53
ILUSTRACIÓN 28. SUBDIRECTORIO DE ARCHIVOS PERTENECIENTES A LAS VISTAS MODELOS.....	55
ILUSTRACIÓN 29. FORMULARIO PARA CREAR PLANIFICACIONES.....	56
ILUSTRACIÓN 30. NUEVA SECCIÓN PARA LOS ELEMENTOS DEL <i>TOOLKIT</i>	57
ILUSTRACIÓN 31. ELEMENTOS DEL <i>TOOLKIT</i>	57
ILUSTRACIÓN 32. CAJA DE HERRAMIENTAS DEL <i>TOOLKIT</i>	58

ILUSTRACIÓN 33. ASPECTO POR DEFECTO DEL ELEMENTO <i>DATEPICKER</i>	59
ILUSTRACIÓN 34. CREACIÓN DE LA CARPETA <i>TOOLKIT.CONTENT</i>	59
ILUSTRACIÓN 35. AGREGAR IMÁGENES DE LOS ICONOS.	60
ILUSTRACIÓN 36. PROPIEDAD ACCIÓN DE COMPILACIÓN.	60
ILUSTRACIÓN 37. TECLADO CON AYUDA EN LA ESCRITURA.....	64
ILUSTRACIÓN 38. ACTIVACIÓN DEL <i>TOMBSTONING</i> AUTOMÁTICO.	68
ILUSTRACIÓN 39. DETALLE DEL ELEMENTO <i>PIVOT</i>	70
ILUSTRACIÓN 40. PRIMERA VISTA DEL <i>PIVOT</i> : ITINERARIO.	71
ILUSTRACIÓN 41. SEGUNDA VISTA DEL <i>PIVOT</i> . FORMULARIO.	72
ILUSTRACIÓN 42. ELEMENTO <i>BING MAPS</i> EN LA CAJA DE HERRAMIENTAS.	73
ILUSTRACIÓN 43. TERCERA VISTA DEL <i>PIVOT</i> : MAPA.....	74
ILUSTRACIÓN 44. AGREGAR REFERENCIA DE SERVICIO.	80
ILUSTRACIÓN 45. CONFIGURACIÓN DE UNA REFERENCIA DE SERVICIO.....	80
ILUSTRACIÓN 46. CONFIGURACIÓN AVANZADA DE UNA REFERENCIA DE SERVICIO.	81
ILUSTRACIÓN 47. DATOS DE NUESTRA NUEVA CUENTA.	82
ILUSTRACIÓN 48. REGISTRO DE NUESTRA APLICACIÓN.	82
ILUSTRACIÓN 49. FALTA DE CREDENCIALES PARA EL USO DEL ELEMENTO <i>BING MAPS</i>	83
ILUSTRACIÓN 50. PÁGINA <i>EDITAPLAN.XAML</i> Y <i>DESCRIPCION.XAML</i>	85
ILUSTRACIÓN 51. PÁGINA <i>MAINPAGE.XAML</i> Y <i>VIAJES.XAML</i>	86
ILUSTRACIÓN 52. VISTA GASTOS.	89
ILUSTRACIÓN 53. ACCESO A LA CÁMARA DEL SISTEMA.....	91
ILUSTRACIÓN 54. VISTA GASTOS.	92
ILUSTRACIÓN 55. VISTA GASTOS DESPUÉS DE HABER INSERTADO VARIOS GASTOS.	93
ILUSTRACIÓN 56. VISTA GASTOS HABIENDO SUPERADO EL 80% DEL PRESUPUESTO MÁXIMO.	93
ILUSTRACIÓN 57. VISTA GASTOS HABIENDO SUPERADO EL PRESUPUESTO MÁXIMO.....	94
ILUSTRACIÓN 58. <i>LISTPICKER</i> EN MODO EXPANDIDO Y EN PANTALLA COMPLETA.	97
ILUSTRACIÓN 59. LISTA DE GASTOS ANTES Y DESPUÉS DE AGREGAR ELEMENTOS.	100
ILUSTRACIÓN 60. LISTA DE GASTOS.	102
ILUSTRACIÓN 61. VISTA NAVEGACIÓN CON LA BRÚJULA HABILITADA.	105

Índice de tablas

TABLA 1. TECLAS DE ACCESO RÁPIDO DEL EMULADOR.	25
TABLA 2. COLORES DEL SISTEMA.	33
TABLA 3. ESTILOS DE BROCHA DEL SISTEMA.	33
TABLA 4. FUENTES DEL SISTEMA	33
TABLA 5. TAMAÑOS DE FUENTE DEL SISTEMA	33
TABLA 6. EJEMPLO DE TABLA VIAJES.	41
TABLA 7. EJEMPLO DE TABLA DIAS	42
TABLA 8. EJEMPLO DE TABLA GASTOS.	42

1.INTRODUCCIÓN

I. Introducción

La llegada al mercado de los sistemas operativos móviles de Android, iOS y en menor medida Blackberry supuso una auténtica revolución, ya que de su mano vinieron multitud de aplicaciones para sacar el máximo provecho a los actuales móviles inteligentes (smartphones). Como en otros negocios, era de extrañar que este gran mercado estuviera dominado tan solo por dos compañías por lo que poco a poco están apareciendo otros sistemas operativos. Es aquí donde entra Windows Phone, el sistema operativo móvil de Microsoft. En este proyecto se intentará explicar su funcionamiento y características de manera clara y sencilla mientras desarrollamos, así resultará más fácil su aprendizaje.

Antes de ponerse a desarrollar la aplicación ha habido que familiarizarse con el sistema operativo, saber cómo interactúa con el usuario, cómo hace uso de la memoria cuando hay una aplicación en ejecución, conocer la arquitectura hardware, etc. Tras todo esto se decidió que la mejor manera de explicar su funcionamiento era con una aplicación en la que se interactuara con las principales funcionalidades del sistema operativo.

Finalmente se decidió publicar la aplicación desarrollada en el Marketplace de Microsoft, de esta manera se cierra el ciclo de vida de desarrollo de una aplicación. Desde que se plantea su programación hasta que se publica para que todo el mundo pueda descargársela.

La aplicación constará de dos partes bien diferenciadas. Una parte estará destinada a la planificación del viaje, en ella el usuario podrá anotar todos los datos relativos a la preparación y planificación de un viaje. Así, podrá establecer un presupuesto máximo, hacer una programación con el itinerario y echar un vistazo al mapa de la zona a visitar para ir familiarizándose con ella.

La segunda parte de la aplicación estará destinada a un uso durante el propio viaje. De forma que el usuario podrá anotar todos los desembolsos realizados para llevar un control del gasto, informándole en todo momento del dinero que le quede hasta llegar al presupuesto máximo que estableció. En estos desembolsos se pueden incluir tanto el importe de los billetes de avión, como los gastos en comida o las diferentes compras realizadas en el viaje.

Además, en esta parte de la aplicación se le proporcionarán al usuario una serie de herramientas para ayudarle en su ubicación. Mostraremos el uso del giroscopio que llevan incorporados la mayoría de dispositivos con Windows Phone para incluir una brújula digital, simulando las brújulas magnéticas de mano. La otra herramienta que ayudará en la ubicación del usuario y veremos cómo interaccionar con ella, será el GPS del dispositivo, que nos servirá para calcular rutas con indicaciones de cómo llegar entre dos puntos.

II. Objetivos

El objetivo del proyecto será enseñar a crear una aplicación para quien desee adentrarse en la programación de aplicación en Windows Phone. Para ello, la aplicación que desarrollaremos a modo de ejemplo será una agenda de viajes. A la par que se desarrolla la aplicación daremos a conocer el sistema operativo. Al ser todavía un sistema operativo minoritario pretendemos detallar en este proyecto cómo es el entorno de programación, y a la par que vamos creando nuestra aplicación, explicar las distintas funciones que tenemos disponibles para su programación.

En base a ese objetivo principal, se proponen los siguientes objetivos parciales:

- Conocer las características del sistema operativo Windows Phone 7.5.

- Familiarizarse con el entorno de desarrollo y las herramientas que facilita Microsoft.
- Optimizar el código todo lo que sea posible, ya que los recursos de los teléfonos móviles son más limitados que los de ordenadores de sobremesa.
- Conocer y hacer uso de las APIs exclusivas de Windows Phone.
- Publicar una aplicación en el Marketplace.

III. Fases del desarrollo

Como en todo proyecto de programación, esta tarea será la final. Para llegar a ella primero debemos superar las siguientes fases:

- **Estado del arte**

Como en el mercado existen varios sistemas operativos móviles, se estudiaron las distintas alternativas que existían y qué se podía ofrecer de cada una. Se buscó un sistema que todavía no estuviese muy popularizado para poder ofrecer mediante este proyecto una ayuda a quien en un futuro se interese en desarrollar en Windows Phone.

- **Introducción al sistema operativo**

Una vez elegido el sistema operativo con el que trabajar, se recopiló información para conocer las principales características: las ventajas y desventajas que ofrecían estas para el desarrollo de aplicaciones. Esta fase también sirvió para familiarizarnos con el entorno de trabajo que ofrece Microsoft al programador.

- **Diseño de aplicación**

En esta fase se diseña la aplicación. Para ello se vieron varias aplicaciones, tanto del sistema operativo con el que trabajamos como de otros existentes en el mercado. Se decidió crear una agenda de viajes ya que en ella se podían integrar varias funcionalidades:

- almacenamiento de datos en bases de datos
- la interacción con el hardware del teléfono, como por ejemplo la cámara de fotos o el GPS
- persistencia de datos
- navegación entre páginas

- **Desarrollo de aplicación**

Teniendo el diseño de la aplicación llegamos a la fase en la que usamos el lenguaje de programación para implementar las distintas funcionalidades.

- **Publicación de aplicación**

Tras finalizar el desarrollo, pasamos a una fase en la que sometemos a distintas pruebas de rendimiento y certificación a la aplicación para poder publicarla en el Marketplace.

- **Documentación del trabajo realizado**

Una vez finalizado todo el proceso de estudio y desarrollo, documentamos el trabajo realizado, exponiendo ejemplos para su comprensión y las distintas decisiones tomadas durante la implementación de las funcionalidades.

IV. Medios empleados

Para la realización de este trabajo, al principio solo se contaba con un ordenador personal. En él se instalaron las herramientas que proporciona Microsoft para el desarrollo de aplicaciones. Para la búsqueda de información se hizo uso de la conexión a internet y de la consulta de libros especializados en distintas bibliotecas.

Posteriormente se adquirió un terminal móvil con el sistema Windows Phone 7.5 instalado para familiarizarnos con todo el entorno visual y ver cómo otras aplicaciones ya publicadas usan las características del teléfono.

V. Estructura de la memoria

Este trabajo está dividido en cuatro grandes bloques. El primero de ellos, contenido en el capítulo 2, consistirá en ver el estado del arte de los actuales sistemas operativos móviles.

En el capítulo 3, daremos a conocer el sistema operativo para el que vamos a crear la aplicación: la arquitectura del sistema, el modelo software, el entorno de desarrollo, etc.

Posteriormente, en el capítulo 4, explicaremos cómo hemos creado la aplicación desde cero. Cada vez que se utilice una característica o un elemento nuevo, se explicará detalladamente su uso y las diferentes opciones de implementación que ofrece. Al mismo tiempo se tratará de hacer un desarrollo lo más eficaz posible usando el patrón de diseño que recomienda Microsoft.

Una vez acabada la explicación del desarrollo de las distintas pantallas que consta la aplicación, en el capítulo 5, se detallará cómo publicarla en el Marketplace para que todos los usuarios de Windows Phone puedan acceder a ella y descargársela en su dispositivo.

La memoria termina con las conclusiones finales del proyecto, en el capítulo 6, más un desglose del presupuesto en el capítulo 7.

2. ESTADO DEL ARTE

Los *smartphones* y *tablets* están en lo más alto de las listas de ventas de aparatos electrónicos. Pero se trata de un mercado muy dinámico en comparación con los otros productos de electrónica de consumo. Aquí las novedades aparecen todos los meses y en cuestión de dos años el panorama de los sistemas operativos móviles ha podido cambiar completamente. Además cada sistema operativo móvil viene acompañado de su propio espacio donde comprar aplicaciones lo que crea una importante oportunidad para que los desarrolladores puedan hacer llegar sus aplicaciones al máximo número de clientes.

Actualmente el mercado lo dominan dos compañías: Goggle y Apple. Entre ambos se llevan entre el 85% y el 90% de la cuota de mercado. Entre estos dos gigantes de la tecnología el ganador es *Google*, con su sistema operativo Android.

I. Android

La presentación de Android fue en 2007, pero no fue hasta octubre de 2008 cuando se produjo la venta del primer teléfono móvil con este sistema operativo instalado. Está basado en una versión modificada de *Linux* 2.6 y la mayor parte del código fue publicada bajo licencia de Apache. La estructura del sistema operativo Android se compone de aplicaciones escritas en lenguaje Java que se ejecutan en un *framework*. Este *framework* o marco de aplicaciones está formado por todas las clases y servicios que utilizan directamente las aplicaciones para realizar sus funciones. La mayoría de los componentes de este *framework* son bibliotecas Java que acceden a recursos a través de la máquina virtual Dalvik.

A pesar de estar en lo más alto del mercado se pueden ver ciertas deficiencias en la plataforma. Por un lado tenemos la posición privilegiada de *Samsung* en lo alto del mercado de fabricantes de móviles, hecho que podría incitar a otros fabricantes a apostar más en serio por otras alternativas. Por otro lado el mercado sigue a la espera de ver qué ocurre con *Motorola* tras su adquisición por *Google*. Por último, tras el lanzamiento del *Nexus 4* a un precio por debajo de la competencia, está por ver cómo haya sentado en el resto de fabricantes y podría desencadenar que se desligaran de *Android* para apostar por otros sistemas operativos móviles alternativos.

De todas formas, son factores que están aún por desarrollar y la plataforma cuenta de momento con buenas críticas en la parte técnica. En todo caso, afectaría más bien a sus relaciones con fabricantes y no con los usuarios finales que siguen manteniendo interés en la plataforma.

II. iOS

El otro gigante del que hablábamos se trata de Apple, que lanzó al mercado en 2007 su sistema operativo iOS y se ganó una buena crítica gracias a la calidad de sus múltiples aplicaciones. iOS se deriva de *Mac OS X*, que a su vez está basado en *Darwin BSD*, y por lo tanto es un sistema operativo *Unix*.

iOS cuenta con cuatro capas de abstracción: la capa del núcleo del sistema operativo, la capa de "Servicios Principales", la capa de "Medios" y la capa de "*Cocoa Touch*".

Actualmente iOS tiene aproximadamente un tercio de los usuarios de Android. Su posición en el mercado estadounidense y el japonés es mucho más fuerte que en Europa, donde Android es el más fuerte. También le está afectando la dificultad para despegar en mercados emergentes debido a su alto precio. Al ser iOS un sistema exclusivo de terminales Apple, su penetración en el mercado dependerá siempre exclusivamente de las ventas de nuevos terminales.

Tras el fallecimiento del fundador y director de la compañía, Steve Jobs y el anunciamiento de la nueva versión del sistema operativo iOS 7, hay que esperar a ver cómo responde el mercado.

III. Blackberry OS

En la actualidad apenas queda entre el 10% y el 15% de cuota de mercado de *smartphones* que no pertenezca a iOS o Android. Por tanto el resto de compañías están haciendo un importante esfuerzo por captar a ese grupo de usuarios y conseguir que su sistema operativo sea rentable.

Aquí entran en escena BlackBerry OS y Windows Phone, con tendencias dispares. Mientras que BlackBerry ha ido perdiendo cuota de mercado mes a mes, el sistema operativo de Microsoft va incrementado esta poco a poco. No obstante la auténtica fortaleza de Windows Phone está en Nokia. A pesar de no ser el único fabricante que crea dispositivos con dicho sistema operativo, su futuro está ligado al fabricante finlandés y más después del sonado acuerdo de colaboración que alcanzaron ambas partes.

El sistema operativo de la compañía anteriormente conocida como RIM está claramente orientado a su uso profesional como gestor de correo electrónico y agenda. Desde la cuarta versión se puede sincronizar el dispositivo con el correo electrónico, el calendario, tareas, notas y contactos de *Microsoft Exchange Server* además es compatible también con *Lotus Notes* y *Novell GroupWise*. BlackBerry Enterprise Server (BES) proporciona el acceso y organización del email a grandes compañías identificando a cada usuario con un único BlackBerry PIN. Los usuarios más pequeños cuentan con el software BlackBerry Internet Service, programa más sencillo que proporciona acceso a Internet y a correo POP3 / IMAP / *Outlook Web Access* sin tener que usar BES.

En la actualidad acaba de presentar la muy esperada nueva iteración de su sistema operativo BlackBerry OS 10 junto a sus nuevos terminales. Esta supone prácticamente una apuesta a todo o nada ya que si estas novedades no tienen una buena acogida entre los usuarios, posiblemente significará la desaparición de una compañía que en su día estuvo en lo más alto de este mercado.

IV. Windows Phone

Por otro lado tenemos el sistema operativo que vamos a tratar en este proyecto. Se trata de Windows Phone 7, un sistema operativo móvil desarrollado por Microsoft, como sucesor del sistema operativo Windows Mobile. Fue presentado en el Mobile World Congress de Barcelona el 15 de Febrero de 2010 con el objetivo de ser una alternativa real a los sistemas operativos móviles más asentados en el mercado como son iOS, Android y Blackberry.

Una de las novedades más característica es la nueva interfaz de usuario, denominada “Metro”. Se basa en unos simples pero efectivos mosaicos dinámicos que muestran información útil al usuario de manera limpia y ligera. Así la navegación por el sistema es mucho más fluida. También se estrena un nuevo concepto llamado HUB, un lugar donde centralizar acciones y agrupar aplicaciones por actividad. En el sistema operativo se pueden encontrar el HUB de imágenes, de contactos, de Office o de Xbox Live. Por ejemplo el HUB de contactos es más que una libreta de direcciones, también muestra las últimas publicaciones, tweets y fotos de amigos.

Al cabo de un año, otra vez en el Mobile World Congress, fue presentada la primera actualización del sistema operativo, Windows Phone 7.5, más conocida como “Mango”. Esta actualización traía importantes mejoras en el rendimiento general, así como la inclusión de copiar y pegar, el navegador Internet Explorer 9 con su soporte para HTML5, multitarea en aplicaciones de terceros e integración con Xbox 360 y Kinect (otros productos de ocio de Microsoft). En septiembre de 2011 Microsoft pone a disposición de los desarrolladores y de manera gratuita el Kit de desarrollo de software Windows Phone SDK 7.1, el cual ofrece todas las herramientas que necesita para desarrollar aplicaciones y juegos para dispositivos con Windows Phone 7.0 y Windows Phone 7.5.

La última actualización recibida por este sistema operativo ha sido la versión 7.8, la cual se lanzó para aquellos dispositivos con Windows Phone 7 que no puedan actualizarse a Windows Phone 8. La nueva versión traerá pequeñas mejoras de rendimiento y sobre todo una nueva interfaz de usuario similar a la versión 8.

Entre las herramientas proporcionadas por Microsoft para el desarrollo de aplicaciones, destaca Visual Studio 2010 Express for Windows Phone. Es el programa de desarrollo integrado (IDE) que proporciona Microsoft y que usaremos principalmente para realizar la aplicación. En este programa encontramos lo necesario para construir nuestra aplicación de manera eficaz pudiendo elegir como lenguaje de desarrollo C# o Visual Basic.

Una de las principales características es que nos permite editar el código XAML directamente en el emulador de la interfaz de dispositivo, por lo que nos ahorraremos escribir mucho código y rápidamente tendremos una idea de cómo quedaría el resultado final.

V. Firefox OS y Ubuntu Phone OS

Durante este año se espera que lleguen nuevos sistemas operativos móviles con sus correspondientes filosofías de productos y apoyos tecnológicos. Firefox OS y Ubuntu Phone OS son algunos de los nuevos sistemas operativos móviles que sustituirán a otros que han desaparecido años atrás como Symbian, Bada (en este caso hay noticias de que se vuelva a relanzar) o Meego.

La novedad con la que cuentan a priori Firefox OS y Ubuntu Phone OS que podrán instalarse en terminales que ya están en el mercado, lo que significa que por cada usuario que instale alguno de estos sistemas operativos móviles no solo ganan un usuario en cuota de mercado sino que, además, se la robará a otro sistema operativo, mayoritariamente Android. Sin embargo esto nos llevará posiblemente a una guerra de cifras, puesto que será más difícil controlar la penetración en el mercado. Hasta ahora prácticamente bastaba con contar el número de unidades vendidas para saber cuántos usuarios tenía un sistema operativo móvil. Ahora esto no será 100% fiable y cada analista o compañía puede tomar datos para conocer la realidad exacta: descargas del sistema desde la web de los desarrolladores, porcentaje de uso en navegación web de sitios populares...

Estamos por tanto, en un momento de cambio en el mercado de los sistemas operativos móviles. La llegada de los nuevos sistemas operativos supondrá un amplio abanico de opciones que anteriormente era más limitado. Aun así, no parece que a corto plazo se vayan a mover de los dos primeros puestos las dos grandes plataformas que están actualmente. Lo que sí es seguro, es que el hecho de que aparezcan buenas alternativas hará que tanto Android como iOS sigan renovándose y ofreciendo novedades con cada actualización para seguir manteniendo una alta cuota de usuarios.

3.INTRODUCCIÓN AL DESARROLLO DE APLICACIONES PARA WINDOWS PHONE

I. Arquitectura de Windows Phone

En el mercado existían tres principales planteamientos diferentes en cuanto al modelo hardware de los distintos sistemas operativos, hasta la llegada de Windows Phone:

- Modelo iPhone: La empresa fabricante, Apple, ejerce un fuerte control sobre el hardware de manera que se encarga de diseñar tanto el sistema operativo iOS como el dispositivo donde se ejecuta. La principal ventaja de este modelo es que el sistema operativo ha sido creado para estar totalmente adaptado al dispositivo y viceversa. Esto conlleva ofrecer la misma experiencia de uso a todos los consumidores. De esta manera la principal desventaja es que la variedad de dispositivos en el mercado es nula, ya que solo existe un producto final cuyo coste de adquisición es bastante elevado ya que se trata de un terminal de alta gama y no todo el mundo puede o quiere pagar.
- Modelo Windows Mobile / Android: Con este modelo tanto Microsoft con Windows Mobile como Google con Android desarrollan sus sistemas operativos de modo que cualquier fabricante puede incluir en su dispositivo sin ningún tipo de restricción de hardware ni en la personalización del interfaz de usuario. La principal ventaja de este modelo sería la cantidad y variedad, ya que el consumidor puede elegir dispositivo de gama baja, media y alta. Sin embargo al haber tanta diferencia del hardware entre un terminal de gama baja con uno de gama alta la experiencia de usuario difiere mucho dependiendo el terminal. Otro problema que nos encontramos con este modelo es que no podemos asegurar al desarrollar una aplicación que vaya a ser compatible con todos los modelos existentes y esto obliga a crear distintas versiones lo que provoca el aumento del coste de desarrollo y actualización.
- Modelo Symbian / Nokia: Este sistema operativo fue el precursor del modelo visto en Windows Mobile y Android, puesto que nació gracias a la unión de varias compañías de móviles encabezadas por Nokia y por tanto existen multitud de terminales. Las principales ventajas de este sistema operativo es la optimización de los recursos, la multitarea y una excelente calidad en llamadas de voz, todo esto valiéndose de un hardware muy limitado. Gracias a esto logro la mayor cuota de mercado hasta 2010, pero Nokia tardó mucho tiempo en liberar el código y permitir a los desarrolladores crear nuevas aplicaciones lo que provoco que tanto desarrolladores como usuarios perdieran interés en este sistema operativo.
- Modelo Windows Phone: Es una mezcla de las dos primeras arquitecturas, ya que Microsoft tenía la experiencia de Windows Mobile y la fragmentación e inconsistencia del sistema a través de distintos dispositivos, pero no querían atarse a un solo dispositivo, con este planteamiento, para poder asegurar la misma experiencia a todos los usuarios Microsoft requiere que todo terminal que desee ejecutar Windows Phone tenga unas características mínimas. En este modelo todos los usuarios obtienen la misma experiencia de uso y los desarrolladores saben que la aplicación funcionará de forma idéntica en todos los dispositivos Windows Phone 7.5 y no estamos encerrados a un solo hardware, ya que existen varios dispositivos de distintos fabricantes que cumplen los requisitos de Microsoft.

Con este modelo en mente, Microsoft creó unas especificaciones mínimas iniciales, llamadas *Chassis 1*, por las que todo fabricante que deseara crear terminales para Windows Phone 7.5 debe guiarse (Ilustración 1):



Ilustración 1. Especificaciones Chassis I

- **Procesador:** ARMv7 Cortex / Scorpion a 1 Ghz.
- **Procesador gráfico:** Soporte hardware completo de DirectX9.
- **Memoria:** 256 Gb RAM y 8 Gb ROM.
- **Sensores:** A-GPS, acelerómetro, iluminación, proximidad.
- **Cámara:** 5 megapíxeles con flash y botón físico de disparo.
- **Multimedia:** Aceleración de audio y video por hardware.
- **Pantalla:** Capacitiva, resolución 800x480.
- **Botones físicos:** Inicio, Buscar, Atrás.

En cuanto al modelo software Windows Phone está basado en Windows CE 6.0 R3, un sistema mucho más avanzado que el utilizado en Windows Mobile 6.X (Windows CE 5.2). Un ejemplo de ello es que Windows CE 5.2 soportaba 32 procesos en su kernel, mientras que Windows CE 6.0 soporta 32.768.

Ahora la *Shell* y la plataforma de aplicaciones residen en la memoria de usuario mientras que los drivers, el sistema operativo, el sistema de archivos, *networking*, el sistema de *rendering* y gráficos y el sistema de actualizaciones residen en el espacio *kernel*. Estamos hablando de un sistema de 32 bits, con lo que solo puede manejar 4 Gb de memoria: 2 Gb para proceso y 2 Gb para el *kernel*.

En la versión R3 de Windows CE 6.0 se añadió soporte para *Silverlight Mobile*, *Internet Explorer Embedded* y otras tecnologías entre las que cabe destacar el soporte nativo de *Flash Lite*. Aunque esta tecnología no está disponible en Windows Phone 7.5. A continuación se muestra una figura de cómo quedaría el Modelo de software en Windows Phone (Ilustración 2):

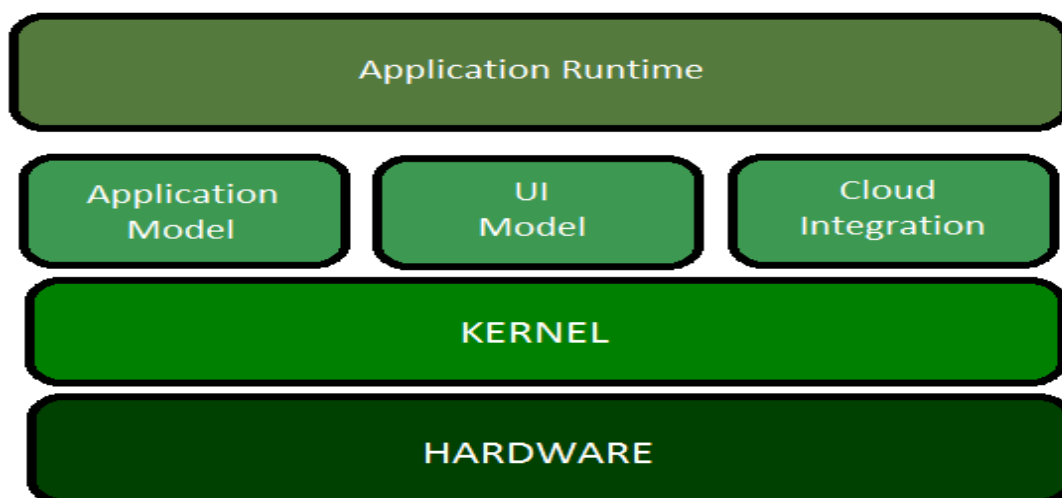


Ilustración 2. Modelo de software en Windows Phone

Modelo de Aplicación (Application Model): En Windows Phone las aplicaciones se despliegan en forma de paquete XAP. Se trata de un archivo comprimido dentro del cual podemos encontrar los ensamblados y recursos originales de nuestra aplicación. La única forma de instalar una aplicación en Windows Phone es mediante la tienda oficial de Microsoft, el Marketplace, en la cual debemos registrarnos como desarrolladores para poder vender nuestras aplicaciones.

Para garantizar la seguridad del sistema y evitar la piratería, el malware o virus, a cada aplicación se le asigne un ID único y un certificado de seguridad emitido cuando nuestra aplicación es aprobada en el Marketplace de Windows Phone.

Modelo de UI (UI Model): El modelo de interfaz de usuario de Windows Phone se compone de elementos, páginas y sesiones. Un elemento es todo control que se muestra al usuario, una página es una agrupación lógica de elementos y una sesión es el conjunto de interacciones que realiza un usuario sobre nuestra aplicación e incluso puede involucrar a otras aplicaciones.

Por ejemplo, podemos realizar una aplicación que necesite del usuario una foto, tenemos un botón (elemento) en nuestra página que abre la galería de imágenes guardadas por el usuario y una vez que se ha seleccionado una, vuelve a nuestra aplicación y se muestra la imagen escogida. Este conjunto de acciones se engloba dentro de una sesión.

Integración con la nube (Cloud Integration): Windows Phone está claramente enfocado a una integración con la nube. Por defecto tenemos integración con servicios como *Exchange*, *Google Mail*, *Hotmail*, *Xbox Live*, *Skydrive*, *Facebook*, *Twitter* o *Bing*. En la versión actual del kit de desarrollo no existen APIs que permitan a nuestras aplicaciones acceder a estos servicios directamente, pero se espera que en próximas actualizaciones si aparezcan.

Runtime de aplicaciones (Application Runtime): El *Application Runtime* nos dice dónde y cómo se ejecutan nuestras aplicaciones, las limitaciones que encontraremos y los *frameworks* que tenemos a nuestro alcance para desarrollar. En la figura se ve cómo queda estructurado el *Application Runtime* de Windows Phone (Ilustración 3):

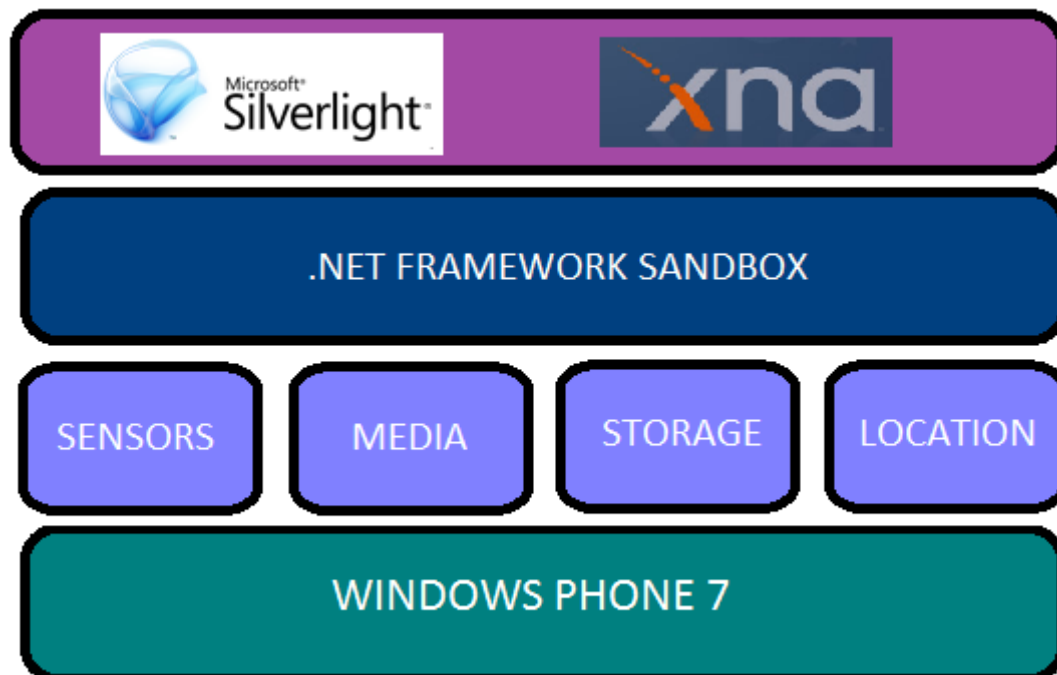


Ilustración 3. Esquema del runtime de aplicaciones de Windows Phone.

En Windows Phone podemos encontrar dos *frameworks* claramente diferenciados: *Silverlight* y *XNA*. Estos *frameworks* se ejecutan sobre un *sandbox* de .NET que les facilita el acceso al hardware, sensores, almacenamiento, localización, etc... Esto quiere decir que nuestras aplicaciones nunca tendrán acceso nativo al sistema y siempre se ejecutarán aisladas del sistema, no pudiendo compartir espacio de almacenamiento ni ningún otro tipo de información a no ser que usemos servicios externos en la nube para ello.

II. Ciclo de vida de una aplicación

Todas las aplicaciones tienen un ciclo de vida, en el cual se inicia, es usada y por último es destruida. Además una aplicación móvil tiene algunas peculiaridades que hay que tener en cuenta. En el caso de Windows Phone sólo podemos tener una aplicación activa y en ejecución al mismo tiempo, debido a esto, nuestra aplicación pasa por diferentes estados, cómo podemos ver en el diagrama a continuación (Ilustración 4):

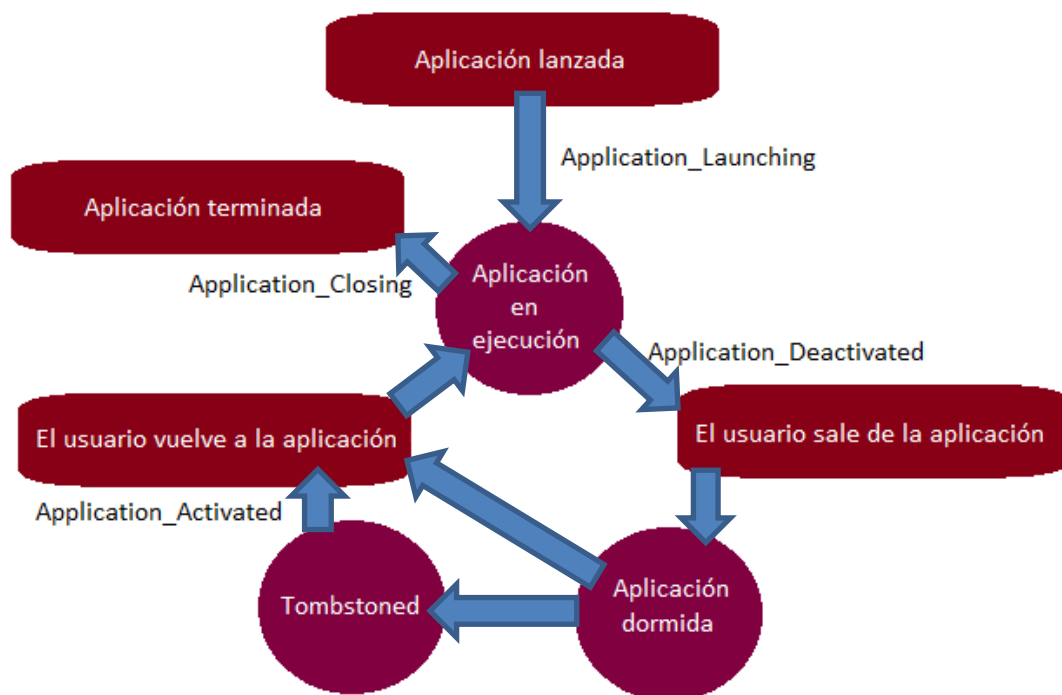


Ilustración 4. Diagrama de vida de una aplicación en Windows Phone

Estos estados se corresponden con eventos que recibe nuestra aplicación y que podemos usar para realizar las acciones necesarias para asegurarnos un correcto funcionamiento.

En primer lugar, cuando nuestra aplicación es lanzada por el usuario, y antes de que la interfaz llegue a mostrarse, recibimos el evento *Application_Launching*.

Tenemos que tener mucho cuidado con el trabajo a realizar en este evento. Las condiciones de aceptación técnica de nuestra aplicación al ser enviada al Marketplace incluyen un apartado específico en el que indican que nuestra aplicación debe mostrar la primera pantalla antes de 5 segundos. De lo contrario el sistema abortará el proceso.

Una vez que nuestra aplicación este en ejecución puede producirse la situación de que el usuario, presionando la tecla inicio, salga sin cerrarla, momento en el cual se lanzaría el evento

Application_Deactivated. Este evento indica que nuestra aplicación está siendo desactivada. Este es el momento en que debemos guardar en el almacenamiento local todos los elementos necesarios para que nuestra aplicación vuelva a iniciarse si el usuario vuelve a ella.

Una vez que la aplicación ha lanzado el evento *Application_Deactivated* esta quedará en estado “durmiente”. Se conservará tal y cómo estaba en el momento de desactivarse, pero no podrá seguir procesando ni ejecutando nada. Si el usuario vuelve a activarla se mostrará en el mismo estado que se encontraba. Pero el estado “durmiente” no es infinito.

Siempre que el dispositivo tenga memoria suficiente para ejecutar aplicaciones, mantendrá las “durmientes”. Pero si se queda sin memoria empezará a realizar el *tombstoning* de las aplicaciones “durmientes”, básicamente cerrándolas y liberando la memoria que ocupaban.

Otra situación que puede disparar la desactivación de nuestra aplicación y el consiguiente lanzamiento del evento *Application_Deactivated* es el uso de lanzadores (aplicación de mapas, el navegador, etc.) y selectores (sacar una foto, obtener un contacto, etc.) en el dispositivo.

Si el usuario vuelve a nuestra aplicación, se lanzará el evento *Application_Activated*. Tanto si se ha mantenido “durmiente” como si se ha realizado el *tombstoning* recibiremos este evento. En este momento podremos recuperar la información que guardamos para ofrecer al usuario la ilusión óptica de que la aplicación ha permanecido en el punto en el cual la abandonó previamente. Para este evento hay que cumplir las mismas exigencias de tiempos de respuesta que para el evento *Application_Launching*.

Tenemos que tener en cuenta la propiedad *IsApplicationInstancePreserved*. Si esta propiedad está establecida en *True*, significa que nuestra aplicación ha estado “durmiente” y no debemos hacer nada para recuperar los datos de nuestra aplicación, ya que es el propio sistema el que se encarga de restaurarla. En caso de que este establecida en *False*, deberemos restaurar los datos que previamente guardamos en el evento *Application_Deactivated*.

Con nuestra aplicación en ejecución nuevamente, el último evento que podremos recibir es el evento *Application_closing*, que se lanzará cuando el usuario este cerrando definitivamente la aplicación.

III. Actualización Windows Phone 7.8

Microsoft ha anunciado el lanzamiento de Windows Phone 8, pero los usuarios existentes, portadores del actual sistema operativo, Windows Phone 7.5 no podrán actualizarse a esa versión. La solución para no dejarles atrás y que estos sigan disfrutando de un sistema operativo a la orden del día, es la actualización a Windows Phone 7.8 que llegó España el 31/01/2013.

Esta actualización incluye una serie de nuevas características que facilitan el uso del teléfono y lo hacen más personalizable.

La primera novedad es que se puede cambiar el tamaño de las ventanas activas (Tiles) entre pequeñas, medianas o grandes y disponer de una pantalla de Inicio totalmente personalizable. Las ventanas activas ofrecen más información, como el texto de los mensajes nuevos o detalles sobre las citas. Las ventanas pequeñas permiten pulsarlas y acceder a ellas con mayor facilidad.

Además se puede animar el teléfono con el doble de colores de énfasis: un mínimo de 20 colores, desde el rojo y el amarillo, hasta el añil y el acero. El color que el usuario elija se aplicará en el inicio y en todos los rincones del teléfono (Ilustración 5):



Ilustración 5. Nueva gama de colores de énfasis

También ha mejorado la pantalla de bloqueo. Cada día cambiará la imagen de Bing a mostrar, siendo en la mayoría de los casos lugares exóticos que ayudarán a mejorar el aspecto general.

Aunque se trata de una actualización sobre todo de diseño, hay que tener en cuenta ciertas restricciones:

- La actualización de Windows Phone 7.8 no está disponible en todos los mercados ni para todos los teléfonos.
- Es posible que algunas características funcionen de manera diferente o que no funcionen en absoluto si el teléfono tiene 256 MB de RAM. Para saber cuánta memoria tiene el teléfono, en Inicio, hay que desplazarse a la izquierda, pulsar en Configuración y, a continuación, pulsar en Acerca de.
- Es posible que determinadas características no estén disponibles en todos los países o regiones.

De cara al desarrollo de futuras aplicaciones esta actualización no ofrece grandes cambios, puesto que como hemos dicho es una actualización básicamente a nivel de presentación. Tendremos que tener en cuenta los nuevos colores para no usar fuentes que sean ilegibles respecto al color del fondo. Además, a la hora de publicar la aplicación tendremos que subir iconos con distintos tamaños cuya resolución sea la adecuada para cada tamaño de ventana activa (Tiles).

IV. Lenguaje de desarrollo

a) XAML: El lenguaje de la presentación

El lenguaje que vamos a utilizar para crear la interfaz de nuestra aplicación es XAML (acrónimo pronunciado *xammel* del inglés *eXtensible Application Markup Language*, Lenguaje Extensible de Formato para Aplicaciones en español). Éste es un lenguaje declarativo, basado en XML y pensado para describir la interfaz gráfica de una aplicación de forma textual y ordenada. XAML es el lenguaje que subyace a la presentación visual de una aplicación desarrollada en Microsoft Expression Blend, al igual que HTML es el lenguaje que subyace a la presentación visual de una página web.

Este lenguaje forma parte de Microsoft Windows Presentation Foundation (WPF). WPF es la categoría de características de Microsoft .NET Framework 3.5 relacionadas con la presentación visual de aplicaciones basadas en Windows y de aplicaciones cliente basadas en exploradores web.

Esto quiere decir que si queremos tener una pantalla en nuestra aplicación con una etiqueta, una caja de texto y dos botones, vamos a poder representar la posición, nombre y estilos escribiendo sentencias XAML que luego, en tiempo de diseño serán interpretadas por el editor que estemos usando (Visual Studio 2010) para mostrarnos la interfaz de usuario y que, una vez la aplicación este en ejecución, serán compiladas a formato binario para que el CLR (siglas de *Common Language Runtime*, un entorno de ejecución para los códigos de los programas que corren sobre la plataforma Microsoft .NET.) pueda analizarlas sintácticamente y recrear un árbol visual de los elementos que componen nuestra interfaz. La sintaxis es XML, un elemento con distintos atributos y el valor entrecomillado de estos atributos.

Una característica muy importante de XAML es que todos los objetos (controles, etc.) que definamos en él, automáticamente son instanciados por el CLR y creados como objetos accesibles desde código, sin necesidad de realizar de nuevo la declaración de los mismos en *code behind* (archivo en el que se da funcionalidad a los objetos de la interfaz gráfica).

Incluso la clase de *code behind* asociada a un archivo XAML es una clase parcial, compartida con el archivo .XAML:

```
<phone:PhoneApplicationPage
    x:Class=WindowsPhoneApplication1.MainPage">
</phone:PhoneApplicationPage>
```

y el archivo .xaml.cs de código:

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }
}
```

Como podemos observar, en la definición de la página en XAML el atributo *x:Class* indica la clase con la que está relacionada nuestra ventana, al ser esta una clase parcial, cuando se compila nuestro archivo XAML se crea una nueva clase parcial automáticamente con todo su contenido declarado. Al pertenecer a la misma clase podemos acceder de forma transparente a los controles y objetos declarados en XAML desde nuestro *code behind*.

b) C#: El lenguaje de la lógica de negocio

Durante los últimos 25 años C y C++ han sido unos de los lenguajes más utilizados para desarrollar aplicaciones comerciales y de negocios. Estos lenguajes proporcionan un altísimo grado de control al programador permitiéndole el uso de punteros y muchas funciones de bajo nivel. Sin embargo cuando se comparan lenguajes como Microsoft Visual Basic con C/C++, uno se da cuenta de que aunque C y C++ son lenguajes mucho más potentes, se necesita mucho más tiempo para desarrollar una aplicación con ellos.

Se necesitaba por tanto un lenguaje que estuviera entre los dos. Un lenguaje que ayudara a desarrollar aplicaciones rápidas pero que también permitiese un gran control.

Facilitar la transición para los programadores de C/C++ existentes y proporcionar a la vez un lenguaje sencillo de aprender para los programadores inexpertos son dos de las grandes ventajas de C#.

Microsoft presentó C# al público en la *Professional Developer's Conference* en Orlando, Florida, en el año 2000. C# combina las mejores ideas de lenguajes como C, C++ y Java con las mejoras de productividad de *.NET Framework* de Microsoft y brinda una experiencia de codificación muy productiva. Microsoft diseñó C# de modo que retuviera toda la sintaxis de C y C++. Sin embargo, la gran ventaja de C# consiste en que sus diseñadores decidieron no hacerlo compatible con los anteriores C y C++. Aunque esto puede parecer un mal asunto, en realidad es una buena noticia. C# elimina las cosas que hacían que fuese difícil trabajar con C/C++. Como todo el código C es también código C++, C++ tenía que mantener todas las rarezas y deficiencias de C. C# parte de cero y sin ningún requisito de compatibilidad, así que puede mantener los puntos fuertes de sus predecesores y descartar sus debilidades.

V. Instalación del entorno de trabajo

a) Requisitos del sistema

Antes de instalar el entorno de desarrollo en el puesto de trabajo tenemos que cerciorarnos de que el equipo cumple una serie de requisitos mínimos:

- Sistema operativo compatible:
 - Windows® Vista® (x86 y x64) con Service Pack 2, todas las ediciones excepto Starter Edition.
 - Windows 7 (x86 y x64), todas las ediciones excepto Starter Edition.
- Para la instalación se requieren 4 GB de espacio de disco disponible en la unidad del sistema.
- 3 GB de RAM.
- Windows Phone Emulator requiere una tarjeta gráfica con funcionalidad DirectX 10 o superior y un controlador WDDM 1.1.

b) Instalación del SDK

Lo primero que tenemos que hacer para empezar a desarrollar en Windows Phone es descargar desde la web de Microsoft y de manera gratuita el Kit de Desarrollo de Software (en adelante nos referiremos a él como SDK, *Software Development Kit*). En él vienen todas las herramientas necesarias:

- **Microsoft Visual Studio 2010 para Windows Phone:** Es la principal herramienta que usaremos para el desarrollo de nuestras aplicaciones.
- **Microsoft Expression Blend para Windows Phone:** Herramienta pensada para diseñadores que nos da control total del aspecto de nuestra aplicación y facilitará el diseño de la interfaz de usuario.
- **Emulador de Windows Phone:** Emulador que nos permitirá probar nuestras aplicaciones en condiciones parecidas a las que podemos encontrar en un dispositivo real.
- **Application Deployment:** Esta herramienta nos permite desplegar aplicaciones en formato XAP (formato en el que se generan los ejecutables de nuestras aplicaciones) al emulador o a un dispositivo, así podremos probar nuestras aplicaciones fuera de Visual Studio.
- **Windows Phone Developer Registration:** Esta herramienta nos permitirá registrar un dispositivo físico Windows Phone con nuestra cuenta de desarrolladores de Marketplace, con lo que el dispositivo quedará desbloqueado para el desarrollo y podremos depurar y desplegar aplicaciones en él.

- **Windows Phone Marketplace Test Tool:** Es un kit de pruebas automáticas y manuales que emulan las pruebas que realiza el Marketplace cuando enviamos una aplicación para su posterior publicación.

Una vez descargado el instalador del SDK 7.1 (esa es la versión del sistema operativo, el producto se llama Windows Phone Mango o Windows Phone 7.5) la instalación es muy sencilla ya que tan sólo ejecutando este archivo se descargan e instalan todos los componentes necesarios.

Finalizado el proceso de instalación del SDK ejecutaremos el Visual Studio 2010 y desde su ventana principal podremos crear un nuevo proyecto pulsando el link “Nuevo Proyecto”.

c) Instalación de paquetes extras

Aunque el SDK de Windows Phone 7.5 incluye la mayoría de elementos necesarios para crear una aplicación con una interfaz de usuario rica y amigable, existen otros elementos más avanzados que nos son útiles.

Para suplir esta falta de elementos, Microsoft u otros desarrolladores publican de forma libre nuevos *frameworks*. Estos *frameworks* contienen elementos listos para ser usados; de los cuales podemos descargar los binarios para usarlos directamente, el código fuente completo y ejemplos de uso y documentación.

Para poder instalar estos *frameworks*, debemos usar NuGet [1]. Este es un gestor de paquetes orientado a los desarrolladores, especialmente pensado para .NET. Se integra con Visual Studio y permite obtener e instalar paquetes de forma sencilla con sentencias de comandos.

Para usar NuGet se accede a <http://www.nuget.org> y presionamos el botón de instalación que nos lleva a la página de NuGet en *Visual Studio Gallery* (Ilustración 6):

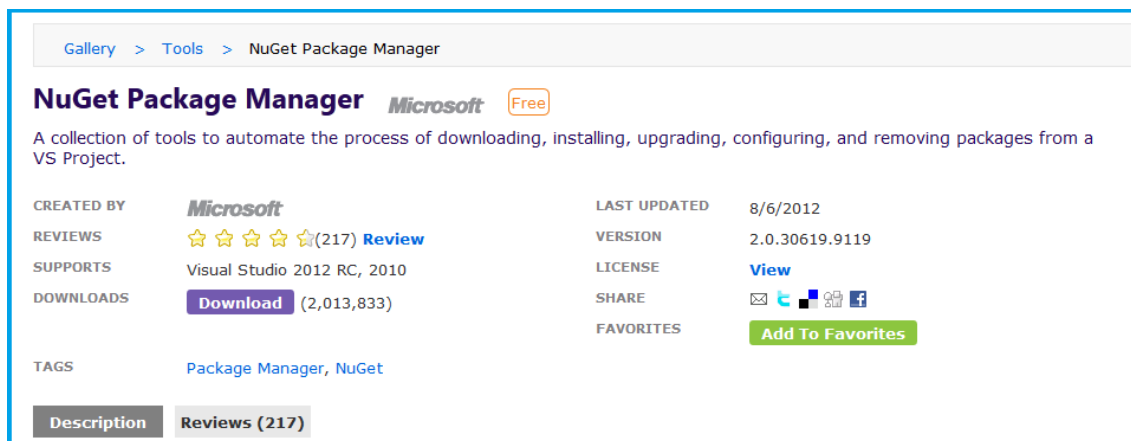


Ilustración 6. Cabecera de la página de descarga de NuGet

Simplemente se presiona el botón *Download* y ejecutamos el archivo de instalación de Visual Studio que se ha descargado. Una vez que lo hayamos instalado, podemos abrir Visual Studio y localizar la consola de administración de paquetes, en el menú Herramientas > Administrador de paquetes de biblioteca > Consola de administración de paquetes y aparecerá la consola en la parte inferior de Visual Studio (Ilustración 7):

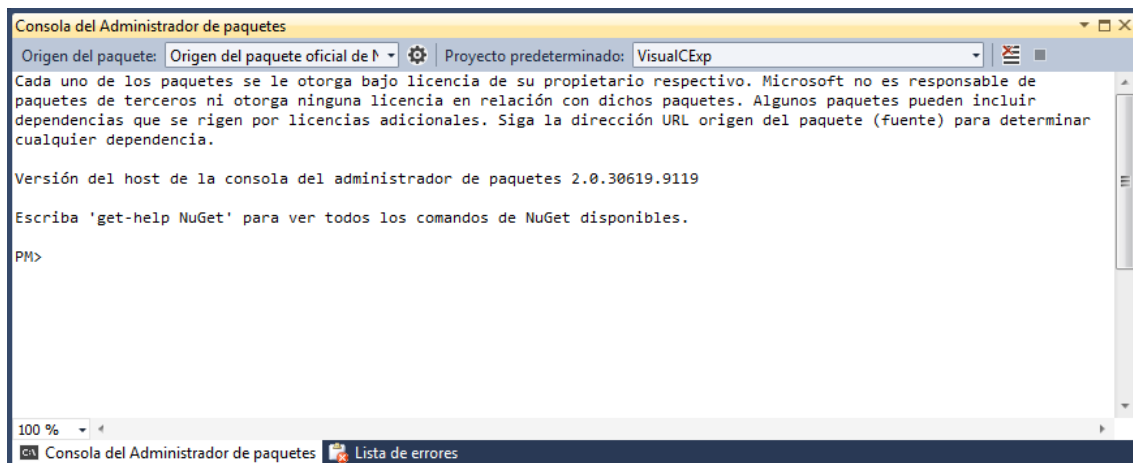


Ilustración 7. Consola de administración de paquetes.

Vamos a instalar el *Toolkit* de Silverlight para poder disponer de nuevos elementos que vamos a necesitar en esta aplicación. En este paquete vienen un total de 18 elementos, pero no vamos a usar todos ellos, sólo aquellos que sean necesarios para mejorar la funcionalidad, como por ejemplo el *DatePicker* o el *ListPicker*. Para comenzar debemos escribir la siguiente línea de comandos:

```
PM> Install-Package SilverlightToolkitWP
```

Y a continuación escribiremos otra línea de código para instalar el paquete *AppBarUtils*, el cual lo usaremos para poder enlazar comandos a los elementos de la barra de aplicación (en el apartado 4.VIII.b explicaremos en detalle esta funcionalidad):

```
PM> Install-Package AppBarUtils
```

Al ejecutar estas sentencias, veremos en la parte inferior de Visual Studio cómo se comienzan a descargar ambos *Toolkits* respectivamente.

Al terminar nos indicará que se ha instalado y también lo referenciará automáticamente en el proyecto en el que estamos trabajando.

De esta forma NuGet nos ofrece una forma rápida y sencilla de instalar componentes de terceros en nuestro sistema y referenciarlos en nuestros proyectos.

4. DESARROLLO DETALLADO DE UNA APLICACIÓN COMPLEJA CON VARIEDAD DE FUNCIONALIDADES

En este apartado vamos a explicar detalladamente cómo desarrollar una aplicación con variedad de funcionalidades en Windows Phone 7.5. Para ello lo primero que se debe hacer es pensar en una aplicación que pueda tener utilidad en un teléfono inteligente (*Smartphone*). Para este proyecto hemos pensado en realizar una agenda de viajes en la que el usuario pueda crear y mantener sus planificaciones de viajes, además de ofrecerle funcionalidades extras durante el viaje. De esta manera tendrá actualizada la organización de sus viajes en todo momento.

I. Empezar a desarrollar una aplicación

Antes de empezar a programar debemos decidir en qué lenguaje queremos desarrollar la aplicación ya que se puede desarrollar tanto en C# como en Visual Basic. En segundo lugar deberemos escoger uno de los *frameworks*; Silverlight o XNA. Aunque los dos se basan en el mismo *framework* de .NET, la diferencia entre ambos es que XNA está pensado para desarrollar aplicaciones que hacen uso intensivo de las capacidades gráficas del dispositivo (principalmente juegos) y Silverlight pensado para otras aplicaciones que usan de forma más general las capacidades multimedia del dispositivo.

A continuación vamos a describir cómo crear una aplicación mientras explicamos más detalladamente el código generado automáticamente y los elementos más básicos. En este caso vamos a crear un sencillo navegador que tendrá tres elementos, un espacio para introducir la página web a la que queremos navegar, un botón de confirmación y un espacio reservado para ver la página web.

Lo primero que tenemos que hacer es abrir el Microsoft Visual Studio 2010 y crear un nuevo proyecto. A continuación se nos mostrará la primera pantalla (Ilustración 8):

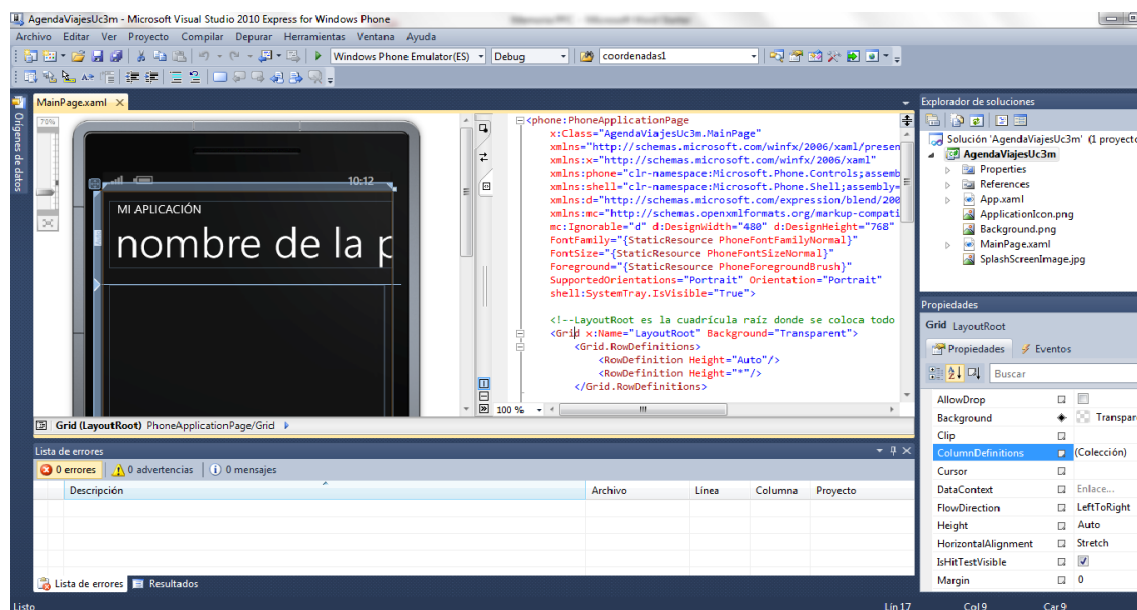


Ilustración 8. Ejemplo de área de trabajo de Visual Studio 2010.

La pantalla se divide en tres columnas, en la izquierda tenemos la presentación de nuestra aplicación en Windows Phone, en el centro tenemos el código XAML que representa la interfaz de la izquierda. Finalmente la columna de la derecha está dividida en dos zonas. En la zona superior tenemos el Explorador de soluciones en que tenemos un acceso al directorio donde se crean todos los archivos del proyecto. En la zona inferior encontramos con una barra de desplazamiento en la que podemos ver las distintas propiedades de cada elemento que seleccionemos, de esta manera podemos modificar cualquier valor y verlo rápidamente en la interfaz.

La estructura del explorador de soluciones es la siguiente (Ilustración 9):

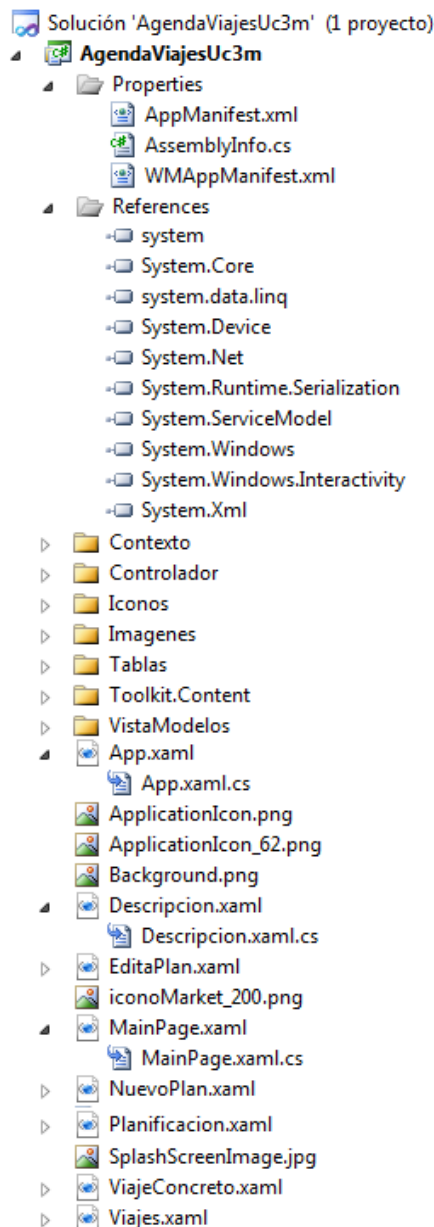


Ilustración 9. Archivos de un proyecto.

Como vemos, todos los archivos se crean dentro del directorio raíz cuyo nombre es el nombre del proyecto:

Properties\AppManifest.xml: Es el manifiesto de la aplicación. Indica las librerías necesarias para crear la aplicación.

Properties\AssemblyInfo.cs: Contiene la configuración del proyecto que se usa al compilar la aplicación. No se debe modificar.

Properties\WMAppManifest.xml: Contiene metadatos con información de la aplicación, título, autor, dirección de los iconos, etc.

References: Carpeta que contiene una lista de las bibliotecas (Ensamblados o Assemblies) que prestan servicios y la funcionalidad que requiere la aplicación para poder trabajar. A lo largo de la aplicación se irán añadiendo más bibliotecas según surjan las necesidades.

App.xaml\App.xaml.cs: Contiene recursos de nivel de aplicación que puede usar cualquier documento dentro de ella. En este archivo podemos definir recursos e introducir código en los eventos del ciclo de vida de la aplicación.

ApplicationIcon.png: Un archivo de imagen que representa el icono de la aplicación en la lista de aplicaciones del teléfono.

Background.png: Es otro archivo de imagen que representa el icono que aparecerá cuando la aplicación es anclada a la pantalla de inicio.

MainPage.xaml\MainPage.xaml.cs: El archivo .xaml es la página principal creada automáticamente. Mientras que el archivo .cs contiene el código subyacente correspondiente a la lógica de la interfaz del archivo MainPage.xaml, en adelante también nos referiremos a él como *code behind*.

El proyecto tendrá tantos ficheros .xaml como páginas tenga la aplicación y por cada archivo .xaml su correspondiente archivo de *code behind* (.cs).

Por otro lado en la carpeta VistaModelos tendremos todos los archivos con código en C# que complementarán la lógica de la aplicación.

SplashScreenImage.jpg: Esta es la primera imagen que se mostrará al iniciar la aplicación. La pantalla de bienvenida da al usuario información inmediata de que la aplicación se está poniendo en marcha y permanecerá encendida hasta que se cargue completamente la primera página.

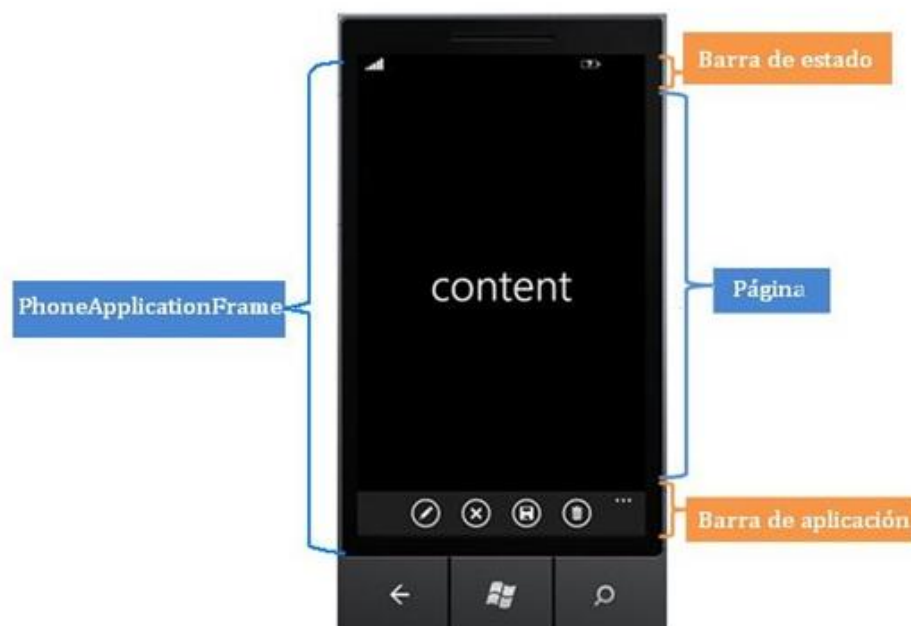


Ilustración 10. Composición del marco en Windows Phone.

La interfaz de una aplicación Windows Phone 7.5 se define usando páginas que contienen los elementos que componen la interfaz. En cada aplicación solo puede existir un control *PhoneApplicationFrame*, que se crea por defecto al abrir un nuevo proyecto. *PhoneApplicationFrame* es el control de navegación principal y admite la navegación hacia y desde las páginas (Ilustración 10).

PhoneApplicationPage encapsula el contenido por el que se puede navegar en *PhoneApplicationFrame*. La unidad básica de trabajo en una pantalla de Windows Phone 7.5 es la página, ésta se define con el objeto *PhoneApplicationPage* seguido de varios atributos que definen su comportamiento y los controles que pueden usar. A continuación mostramos un ejemplo del código de los atributos de la página *MainPage.xaml*, estos atributos suelen ser iguales para todas las páginas de una aplicación.

```
<phone:PhoneApplicationPage
    x:Class="MiniExplorador.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">
```

Anteriormente ya habíamos visto el atributo *x:Class*, que especifica la clase de *code behind* con la que trabajará nuestra página XAML. Ahora también encontramos el atributo *xmlns*, el cual especifica los *namespace* a los que tendremos acceso desde este archivo XAML. Como podemos observar, si nos fijamos en los atributos *xmlns:x* y *xmlns:phone* podemos referenciar tanto ensamblados en nuestro equipo como *namespace* XAML usando su nombre, “:” y el nombre del objetos que queramos usar.

En los atributos *mc:Ignorable*, *d:DesignWidth* y *d:DesignHeight*, establecemos el valor por defecto de una página y asignamos un tamaño estándar. Las siguientes líneas de código están ocupadas por los controles para establecer la familia, el tamaño y la posición de la fuente que queremos usar. Con el control *SupportedOrientations* establecemos el valor *PortraitOrLandscape* para que cuando giremos el teléfono la pantalla se muestre en vertical u horizontal.

En nuestro ejemplo tenemos dos filas, la primera de altura automática, y la segunda rellena el resto del espacio:

```
<Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*/>
    </Grid.RowDefinitions>
```

El control *Grid* nos permite definir un área de pantalla formada por filas (*rows*) y columnas (*columns*). Para definir múltiples filas usamos la colección *RowDefinitions*. Cada elemento *RowDefinition* define una fila única especificando su altura.

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="Mi primer HolaMundo"
        Style="{StaticResource PhoneTextNormalStyle}"/>
    <TextBlock x:Name="PageTitle" Text="Mi explorador" Margin="9,-7,0,0"
        Style="{StaticResource PhoneTextTitle1Style}"
        TextAlignment="Right" />
</StackPanel>
```

El control *StackPanel* nos permite apilar los elementos que introduzcamos en él automáticamente de forma horizontal o vertical, como si de una lista se tratase.

Con los atributos del ejemplo le damos un nombre (*Name*), establecemos donde (*Grid.Row="0"*) queremos que se muestre y asignamos valores a los márgenes (*Margin*).

Dentro del control *StackPanel* tenemos dos elementos *TextBlock* que contienen el nombre de la aplicación y el título de la página.

A continuación vamos a añadir en nuestro código un *TextBox* para poder escribir la URL (sigla en inglés de *uniform resource locator*) a la que queremos acceder. Esta vez la situamos en la segunda fila (*Grid.Row="1"*) y le damos una anchura y altura automática para que se adapte al texto. Cabe destacar la propiedad *HorizontalAlignment="Stretch"* en la que indicamos que el ancho del *TextBox* se adapte al ancho de la fila y en la propiedad *VerticalAlignment="Top"* indicamos que permanezca siempre arriba de ésta:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBox Height="Auto" HorizontalAlignment="Stretch" Margin="0,9,86,0"
Name="TextBox" Text="http://www.uc3m.es" VerticalAlignment="Top" />
    <Button Content="Ir" Height="72" HorizontalAlignment="Right"
Margin="362,9,0,0" Name="Botón" VerticalAlignment="Top" Width="94" />
    <phone:WebBrowser Height="Auto" HorizontalAlignment="Stretch"
Margin="16,79,0,0" Name="Navegador" VerticalAlignment="Stretch" Width="Auto" />
</Grid>
```

También hemos añadido el correspondiente botón de confirmación, para que una vez hayamos escrito la URL, el navegador acceda a ella. Con el atributo *Content* establecemos la palabra que va a contener el botón en la interfaz.

Por último tenemos el código correspondiente al navegador, en el que usamos las propiedades *HorizontalAlignment="Stretch"* y *VerticalAlignment="Stretch"* para que ocupe el resto del espacio disponible en la página. Hay que reseñar que debemos otorgar un nombre a los elementos a los que queramos acceder posteriormente desde código, en este caso al elemento *WebBrowser* le llamamos Navegador.

Solo faltaría darle la funcionalidad al botón "Ir" para que cuando escribamos una dirección en el *TextBox*, el navegador nos dirija a ella. Este tipo de funcionalidades las vamos a definir en el fichero *MainPage.cs*.

Para ello añadimos el evento *Click* al botón con un manejador llamado *button1_Click*. Este manejador estará ubicado en la clase principal, perteneciente al fichero *MainPage.xaml.cs*, y dentro de él escribiremos la funcionalidad que queramos. En este caso llamamos al método *Navigate*, de la clase *WebBrowser*, cuyo parámetro de entrada es un nuevo objeto de la clase *Uri*. Este objeto representa una URI y necesita de dos parámetros para ser inicializada. El primero de ellos (*TextBox.Text*) lo obtenemos del *TextBox* de la interfaz gráfica, usando el método *Text* para acceder al contenido que introdujo el usuario. El segundo de los atributos define el tipo de URI, que en este caso será absoluta.

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
    }
    private void button1_Click(object sender, RoutedEventArgs e)
    {
        Navegador.Navigate(New Uri(TextBox.Text, UriKind.Absolute));
    }
}
```

Una vez finalizado el código, es momento de pulsar F5 para arrancar el depurador y si no hemos cometido errores en pocos segundos se mostrara en el emulador la aplicación que acabamos de realizar con toda la funcionalidad que hemos añadido (Ilustración 11).

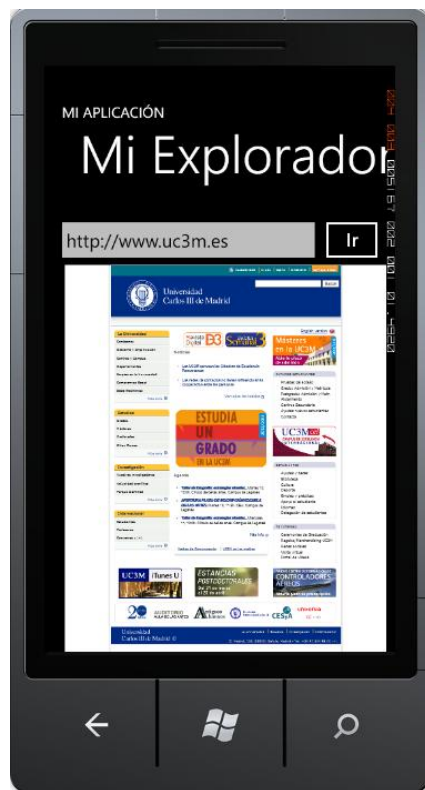


Ilustración 11. Aspecto final de la aplicación.

a) Emulador de Windows Phone

Junto con Visual Studio 2010 el kit de desarrollo de Windows Phone también incluye un emulador totalmente funcional del sistema, que nos permitirá probar nuestras aplicaciones en condiciones parecidas a las que podemos encontrar en un dispositivo real.

Podremos lanzar el emulador en Windows desde Inicio > Programas > Windows Phone Developer Tools > Emulador de Windows Phone.

Podemos indicarle a Visual Studio que deseamos usar el emulador para ejecutar nuestra aplicación en vez del dispositivo real, desde el combo situado a la derecha del botón *Play* (inicio de depuración) que encontraremos en la barra estándar de herramientas (Ilustración 12):

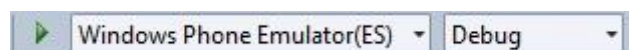


Ilustración 12. Selección del destino de ejecución.

Aunque nunca reemplazará las pruebas en un dispositivo real, el emulador de Windows Phone nos va a permitir de forma sencilla probar incluso aplicaciones que dependan de la localización o que usen los sensores, como el acelerómetro del teléfono.

En la barra de botones del emulador, el último de ellos (>>) nos permite acceder a la pantalla de herramientas adicionales (Ilustración 13):



Ilustración 13. Acceso a las herramientas adicionales del emulador.

En estas herramientas tendremos acceso a un simulador GPS que nos permitirá crear rutas y puntos que se enviarán a nuestro emulador para que nuestra aplicación pueda usarlos (Ilustración 14):

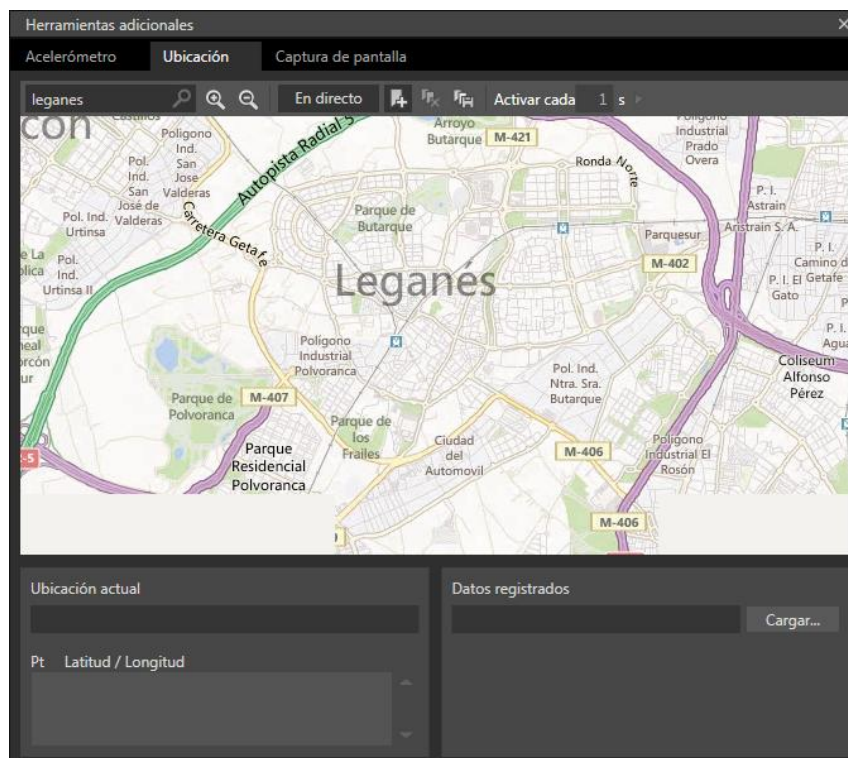


Ilustración 14. Emulador de GPS enviando posiciones a nuestra aplicación.

También tenemos una herramienta que nos permitirá emular el comportamiento del acelerómetro en nuestra aplicación, con un modelo 3D del teléfono que podremos girar como deseemos (Ilustración 15):

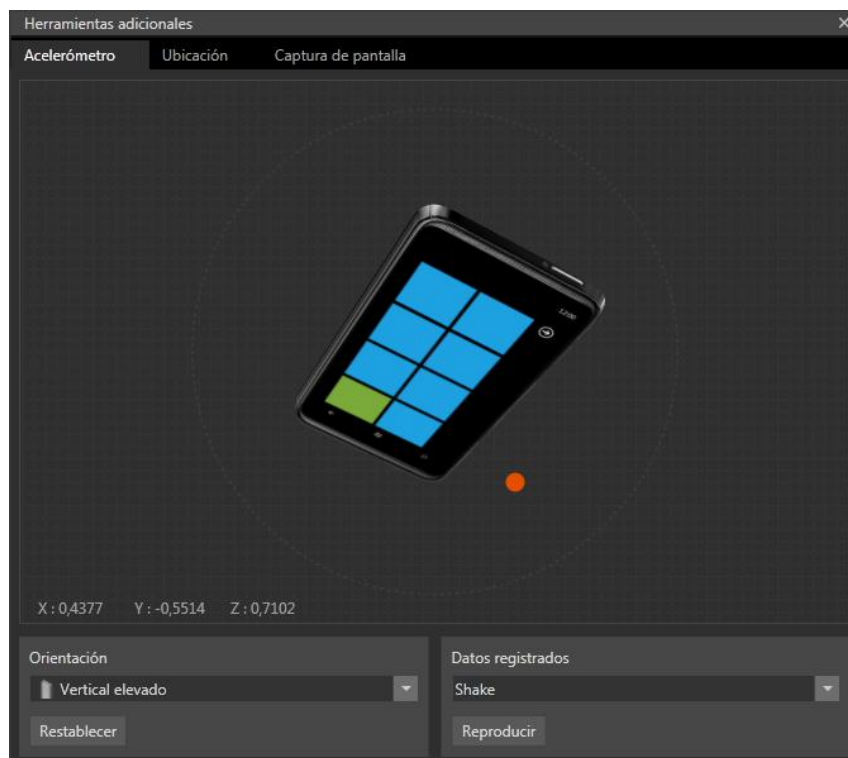


Ilustración 15. Envío de datos de acelerómetro a nuestro emulador.

El emulador de Windows Phone nos permite usar combinaciones de teclas para acceder a funcionalidades del teléfono. A continuación podemos ver una tabla con todas ellas (Tabla 1):

Tecla	Acción
Retroceder Página (Page Up)	Habilita el teclado en el emulador
Avanzar Página (Page Down)	Deshabilita el teclado en el emulador
F1	Botón físico Atrás
F2	Botón físico Inicio
F3	Botón físico Búsqueda
F6	Botón cámara Media pulsación
F7	Botón cámara pulsación completa
F9	Botón Subir volumen
F10	Botón Bajar volumen

Tabla 1. Teclas de acceso rápido del emulador.

b) Application Deployment Tool

Esta herramienta se instala junto al resto a la par que instalamos el SDK de Windows Phone. Para ejecutarla simplemente debemos buscarla en su ruta de instalación, C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\XAP Deployment\ XapDeploy.exe (ésta puede variar dependiendo del equipo). Nos permitirá desplegar aplicaciones en formato XAP al emulador o a un dispositivo desbloqueado para desarrollo. Los ficheros con formato XAP son usados para distribuir e instalar aplicaciones software en Windows Phone. Se trata del fichero comprimido que contiene el archivo con el manifiesto de la aplicación y las librerías necesarias para la instalación en los terminales.

El fichero en formato XAP se crea tras compilar el proyecto y se ubica en el subdirectorio del proyecto \AgendaViajesUc3m\Bin\Debug\ o en \AgendaViajesUc3m\Bin\Release\.

De esta forma podremos probar las aplicaciones fuera de Visual Studio. Tendremos la posibilidad de elegir dónde probar la aplicación cambiando la opción “Destino”. Elegiremos “Windows Phone Device” para instalar la aplicación en un dispositivo (para ello tendremos que conectar el teléfono al PC mediante un cable USB y desbloquear la pantalla de inicio) o bien elegiremos “Windows Phone Emulator (ES)” si queremos lanzar la aplicación en el emulador de Windows Phone que incluye el SDK. (Ilustración 16):



Ilustración 16. Herramienta de despliegue de aplicaciones.

Si disponemos del proyecto original de la aplicación podremos hacer lo mismo desde Visual Studio, haciendo clic derecho sobre proyecto y seleccionando la opción “Deploy” (Desplegar). Esta acción instalará la aplicación en el dispositivo seleccionado en la barra de herramientas (El emulador o un dispositivo físico).

c) Developer Registration Tool

Esta herramienta la encontramos en la ruta C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\Phone Registration (la ruta puede variar en función del equipo). Para ejecutarla simplemente tendremos que hacer doble clic en el archivo PhoneReg.exe. La función de esta herramienta es permitirnos registrar un dispositivo físico Windows Phone con nuestra cuenta de desarrolladores de Marketplace. Con lo que este dispositivo quedará desbloqueado para desarrollo y podremos depurar y desplegar aplicaciones en él (Ilustración 17):



Ilustración 17. Herramienta de registro de dispositivos.

Para el correcto funcionamiento de la herramienta, deberemos tener el dispositivo conectado al PC, sincronizado con Zune y en la pantalla inicial de Windows Phone. Si el dispositivo está apagado o con bloqueo de pantalla fallará.

Zune es un software que se instala en el PC y sincroniza archivos multimedia con el teléfono, descarga actualizaciones del mismo y consigue aplicaciones.

Debemos saber que no es necesario que la cuenta de *LiveID* del dispositivo sea la misma que usamos en el Marketplace. En caso de necesitar más, podemos ponernos en contacto con Microsoft directamente, donde estudiarán nuestro caso particular y podrán acceder a habilitar más dispositivos en nuestra cuenta.

II. Patrón MVVM

Antes de entrar en más detalles sobre el desarrollo, habría que explicar un concepto en el que se basa toda aplicación basada en Silverlight. Se trata del patrón de diseño de la capa de presentación MVVM, éstas son las siglas de *Model – View – ViewModel*, en español: Modelo – Vista – Vista Modelo.

Fue dado a conocer el 8 de octubre de 2005 por John Grossman, Arquitecto de Silverlight y WPF ¹ en *Microsoft* [2].

El patrón MVVM nace como una adaptación del patrón PM (*Presentation Model*) [3], el cual a su vez nace como una extensión del patrón MVP (*Model – View - Presenter*).

El objetivo del patrón MVVM es separar de una forma efectiva la visualización (*View*) de nuestra pantalla (página, ventana) de la lógica de visualización (*ViewModel*, código que interactúa y responde ante el

¹ Windows Presentation Foundation (WPF) es un subsistema de presentación unificado para Windows. Unifica la forma en que Windows crea, muestra y manipula documentos, elementos multimedia e interfaces de usuario (UI), lo que permite a programadores y diseñadores crear experiencias de usuario de forma más rápida y eficiente.

usuario) y de toda nuestra lógica de negocio y *backend* (*Model*, nuestro modelo de datos y su lógica adjunta) (Ilustración 18):

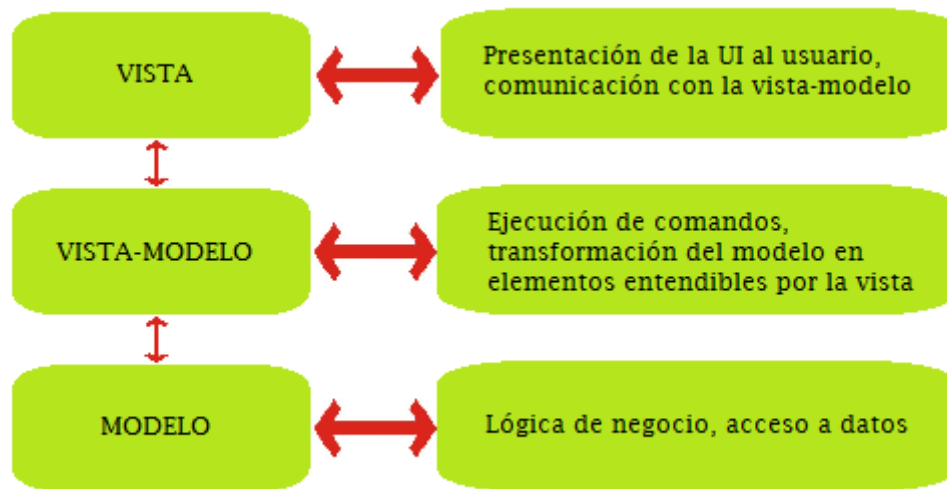


Ilustración 18. Esquema MVVM.

Como podemos ver en este esquema, en MVVM nuestra vista (*View*) no conoce el modelo (*Model*) que está por debajo, solo se relaciona con la vista modelo (*ViewModel*) que se encarga de comunicarse con el modelo para recuperar o persistir información y expone esta información hacia la vista. Nuestra vista modelo también es la encargada de realizar las acciones demandadas por la interfaz, de forma que la vista solo sea una definición de nuestra interfaz, sin lógica ni código, y nuestra vista modelo contenga toda esa lógica totalmente aislada de la representación visual de la misma.

Todo esto se consigue mediante el sistema de enlace a datos de Silverlight. Nuestra vista modelo expondrá los datos que recupere del modelo mediante propiedades con notificación de cambios, a las cuales se enlazarán la vista mediante su propiedad *DataContext* y expresiones de enlace a datos. La propiedad *DataContext* permite que la establezcamos a cualquier objeto al que luego podremos acceder desde expresiones de enlace a datos.

Al desarrollar aplicaciones tendremos que obtener los datos que introduce el usuario. Por ejemplo le presentamos una caja de texto (*TextBox*) para que la rellene con su nombre. El código en XAML sería así:

```
<TextBox Name="txtNombre"
    Text="{Binding Nombre, Mode=TwoWay}"
    Width="480">
</TextBox>
```

Mediante la directiva *Binding* indicamos que ese dato lo queremos recuperar en la vista modelo. Para ello tendremos que crearnos un archivo de código C# donde tendremos la clase de nuestra vista modelo, el constructor y los métodos *get* y *set* para trabajar con nuestro dato:

```
public class VMLogin
{
    private String nombre;

    public VMLogin()
    {
        nombre = "";
    }

    public string Nombre
    {
```

```

        get
        {
            return nombre;
        }
        set
        {
            nombre = value;
            NotifyChanged("Nombre");
            ComandoCrear.RaiseCanExecuteChanged();
        }
    }
}

```

Por último veremos cómo el modelo (archivo Login.xaml.cs) se relaciona con la vista modelo mediante su propiedad *DataContext*:

```

public partial class NuevoPlan : PhoneApplicationPage
{
    VMNuevoPlan contexto;
    this.DataContext = contexto;
}

```

Para ejecutar acciones nos serviremos de un tipo de objeto llamado comando. La ventaja de un comando es que podemos crear una propiedad pública en nuestra vista modelo de tipo *ICommand*, y usar una expresión de enlace a datos para enlazar esta propiedad de nuestra vista modelo con la propiedad *Command* de un botón, por ejemplo:

```

<Button Content="Aceptar" Height="100" Name="botonAceptar"
Width="300" Command="{Binding ConfirmarCambios}"/>

```

En un archivo aparte, *ConfirmarCambios.cs*, contendremos la vista modelo y crearemos la propiedad pública para enlazarla con la propiedad *Command* del botón:

```

public class ConfirmarCambios : ICommand
{
    public PruebaComando () {}

    bool ICommand.CanExecute(object parameter)
    {
        return true;
    }

    public event EventHandler CanExecuteChanged;

    void ICommand.Execute(object parameter)
    {
        // Aquí va el código con la funcionalidad que queramos dar al botón
    }
}

```

Esta clase muestra la infraestructura básica de un comando. Tenemos un constructor de la clase, el método *CanExecute* y el método *Execute* y por último el evento *CanExecuteChanged* (en la sección 4.V.b veremos en profundidad el concepto).

Con estas dos técnicas intentaremos minimizar en lo posible el código en nuestro archivo *code behind*, y no usar eventos, aprovechándonos del enlace a datos para usar propiedades y comandos que tengamos definidos en nuestra vista modelo.

En cuanto al modelo, es un concepto, más que una parte propiamente dicha de nuestra aplicación. Cuando hablamos de modelo, estamos refiriéndonos a nuestras capas de lógica de negocio, acceso a datos, etc. Podemos realizar una aplicación en la que nuestro modelo forme parte de nuestra capa de presentación, o puede ser que exponamos toda nuestra lógica de *backend* a través de servicios web. Esto no significa

que tengamos que tener una capa de modelo en nuestra aplicación que se comunique con esos servicios. Esos servicios son nuestro modelo, y nuestra vista modelo se comunicará directamente con nuestro modelo, ya sea éste clases en nuestra aplicación o un servicio remoto al que accedamos.

III. Estilo y diseño de las páginas

Otro aspecto que tenemos que ver antes de pensar la lógica de negocio de nuestra aplicación es la organización de los distintos elementos de las páginas así como el aspecto que éstas tendrán para que sean más atractivas a la vista del usuario. Aunque no profundizaremos mucho en la explicación de estos términos, ya que para su mejor uso habría que usar el *framework* XNA y no es el objetivo de este proyecto adentrarse mucho en el aspecto visual de la aplicación.

- a. Elementos.
- b. Controles.
 - i. *Canvas*.
 - ii. *StackPanel*.
 - iii. *Grid*.
- c. Recursos.
- d. Estilos.
- e. Plantillas.

a) Elementos

En Silverlight tenemos objetos visuales que componen una pantalla, a los cuales nos referimos como elementos de la misma. Estos son los botones, imágenes, cajas de texto, etc.

Internamente los elementos que tienen apariencia visual en Silverlight heredan de dos clases base: *UIElement* y *FrameworkElement*, que aportan al elemento otras características. Por último sobre estas dos clases bases se construye la clase *Control* (que analizaremos a continuación), que acaba de añadir las propiedades necesarias a nuestros elementos para que sean útiles a la hora de construir la interfaz de nuestra aplicación.

La clase base *UIElement* no expone constructores públicos y normalmente no heredaremos nuestros elementos de ella. Usaremos clases derivadas como *Control*, *ContentControl*, *UserControl*, clases de paneles como *Grid* o directamente heredando de otros elementos ya existentes como *TextBox*, *Button*, etc.

La mayor parte del comportamiento de entrada de un elemento se define en esta clase: eventos para el teclado, entrada con ratón o los eventos de cogida y pérdida del foco.

La clase *FrameworkElement* extiende a *UIElement* añadiendo nuevas características a esta última cómo *layout*, eventos de ciclo de vida (*Loaded*, *Unloaded*) y el soporte para establecer valores de enlace a datos con un contexto heredado.

b) Controles

Una vez que nuestro elemento tiene todas las propiedades de comportamiento, enlace a datos, *layout* y eventos, la clase *Control* acabará de formarlo tal y como llega a nuestra pantalla.

Una de las partes más importantes de la clase *Control* es que define para un elemento la propiedad *Template*, de tipo *ControlTemplate*, y en donde, según veremos más adelante, se define toda la apariencia visual de un elemento. De esta forma es posible reescribir la apariencia visual de un control estándar sin perder la funcionalidad que nos ofrece.

Antes de ver los tipos de controles que sirven de panel y sus características hay que explicar una parte importante del funcionamiento visual de Silverlight: las propiedades adjuntas.

Una propiedad adjunta es un concepto definido por XAML por medio del cual los elementos contenidos dentro de otro elemento pueden heredar y especificar estas propiedades. De esta forma, un control *Canvas* expone las propiedades adjuntas *Top*, *Left* y *ZIndex*, por lo que un botón que se encuentre dentro de un *Canvas* podemos acceder a estas propiedades y especificarlas:

```
<Canvas>
    <Button Canvas.Left="20" Canvas.Top="15"></Button>
</Canvas>
```

De esta forma, el elemento (en este caso el botón) informa al *Canvas* su posición deseada, pero es el control que expone las propiedades adjuntas (*Canvas*) el encargado de posicionar el elemento.

i. Canvas

El control *Canvas* es el más simple de todos los controles de posicionamiento de Windows Phone 7.5. Nos permite posicionar elementos dentro de sus área de forma absoluta, usando coordenadas X e Y expresadas en píxeles, normalmente a través de las propiedades adjuntas *Canvas.Top* (distancia entre el elemento y el borde superior del *Canvas*) y *Canvas.Left* (distancia entre el elemento y el borde izquierdo del *Canvas*). También nos permite establecer el orden visual de los objetos con la propiedad adjunta *Canvas.ZIndex*.

ii. StackPanel

El control *StackPanel* permite apilar los elementos que introduzcamos en él automáticamente de forma horizontal o vertical, como si de una lista se tratase.

No podemos especificar la posición de los objetos, se van colocando en el orden en que los introducimos en XAML y su posición viene dada por el tamaño de los elementos que hayamos colocado antes, adaptándose dinámicamente si ésta varía.

Podemos modificar la dirección en la que nuestros elementos se apilan usando la propiedad *Orientation*, que admite los valores Horizontal y Vertical.

iii. Grid

El control *Grid* nos permite definir un área de pantalla formada por filas (*rows*) y columnas (*columns*) con capacidad para adaptarse a los cambios de tamaño de la pantalla, todo esto en conjunto permite realizar interfaces de usuario complejas.

Por defecto una *Grid* contiene una sola columna y fila. Para definir múltiples columnas usamos la colección *ColumnDefinition*, o *RowDefinition* para definir múltiples filas. A cada elemento *ColumnDefinition* o *RowDefinition* se le podrá asignar una anchura o altura diferente.

Podemos combinar la definición de columnas y filas para crear una rejilla (*Grid*) de celdas en las que posicionar elementos. Para indicar a un elemento que se encuentre en nuestra *Grid* en qué celda debe situarse, lo hacemos mediante las propiedades adjuntas *Grid.Row* y *Grid.Column*. De esta forma la celda indicada se convierte en contenedor de nuestro elemento a efectos de margen, alineación horizontal o vertical y otros parámetros.

El tamaño de las columnas o filas de una *Grid* se puede expresar de tres formas diferentes: tamaño fijo, automático o relativo.

Para establecer el tamaño fijo simplemente tenemos que basarnos en píxeles. Pero si nuestra *Grid* cambia su tamaño, no podemos adaptar el contenido de esta al nuevo tamaño. O si el contenido de una celda crece, el resto de celdas no se desplazarán para adaptarse al tamaño del nuevo elemento.

Para hacer que una columna, por ejemplo, se adapte automáticamente al tamaño de su contenido, en su propiedad *Width* podemos especificar “Auto” como valor, al igual que en la propiedad *Height* de una fila.

Para indicar que una fila o columna ocupe un espacio relativo al tamaño total de la *Grid* usaremos el asterisco, que seguido de un valor indicará el porcentaje del total a ocupar.

c) Recursos

Para mantener un aspecto uniforme en todas las páginas de nuestra aplicación vamos a usar los recursos. En Silverlight llamamos recurso a una pieza de XAML que vamos a reaprovechar a lo largo de nuestra aplicación, asignándole una clave única o indicando el tipo de elemento al que se aplica.

Todo elemento tiene una colección de *Resources* dentro de la cual podemos especificar estos recursos. Desde un botón hasta una página, en todos encontraremos esta colección. Además todo proyecto de Silverlight contiene un archivo *app.xaml*. Los recursos que especifiquemos dentro de él estarán accesibles desde cualquier página de la aplicación.

Por ejemplo, queremos tener el mismo fondo en todas las páginas de la aplicación. Una opción sería definirlo en cada página dentro la propiedad *Background* de la *Grid* principal. Pero, ¿qué ocurre si queremos cambiar el color que hemos establecido? Tendríamos que ir página por página cambiando el color. Lo más sencillo sería establecer el color de fondo en un solo lugar. Dentro de los Recursos de la aplicación (archivo *app.xaml*) y luego mediante el enlace a recursos (instrucción *StaticResource*) se indica a todos los elementos que deseen usar ese color cuál es el recurso que deben buscar:

```
<Application.Resources>
    <ImageBrush x:Name="imagenFondo"
ImageSource="/AgendaViajesUc3m;component/Imagenes/fondo_verde.png"></ImageBrush>
</Application.Resources>
```

En este caso en vez de utilizar un color de fondo hemos usado una imagen. En el código XAML llamamos al recurso que hemos creado “imagenFondo”. Ahora solo debemos indicarlo en las *Grids* principales de nuestras páginas:

```
<Grid x:Name="LayoutRoot" Background="{StaticResource imagenFondo}">
```

De esta forma, cuando queramos cambiar el color o la imagen de fondo de nuestra aplicación, solo tenemos que cambiar el valor del recurso “imagenFondo” y automáticamente todas nuestras *Grids* aplicarán el nuevo valor establecido.

La forma de realizar la expresión a la hora de acceder al recurso, encerrando la sentencia entre llaves, indica al *parser* de XAML que no estamos indicando un valor literal, sino una expresión que debe procesar y resolver. *StaticResource* le indica que estamos accediendo a un recurso declarado en algún lugar dentro de nuestra aplicación. Por último usa el nombre del recurso para localizarlo en las colecciones de recursos, obtener el valor y aplicarlo a la propiedad indicada, en este caso *Background*.

En Windows Phone 7.5 es muy importante la apariencia de nuestra aplicación en relación con el resto del sistema, siempre intentando mimetizarla con el diseño Metro de Windows Phone. Para hacernos más sencilla esta tarea tenemos acceso a una serie de recursos que exponen características del dispositivo, como por ejemplo el color de fondo actual (blanco o negro), el color de letras (blanco o negro), el color del resaltado (azul, naranja, rojo...), etc. seleccionado en el dispositivo, fuente de letras usadas y muchos más.

Accedemos a estos recursos de la misma forma que a otros definidos en nuestra aplicación, mediante la expresión *StaticResource* y el nombre del recurso. A continuación vemos unas tablas con los recursos disponibles:

Clave	Valor por defecto
PhoneBackgroundColor	#FF1F1F1F
PhoneContrastForegroundColor	Black
PhoneForegroundColor	White
PhoneInactiveColor	#FF666666
PhoneDisabledColor	#FF808080
PhoneSubtleColor	#FF999999
PhoneContratsBackgroundColor	#FFFFFF
PhoneTextBoxColor	#FFBFBFBF
PhoneBorderColor	#FFCCCC
PhoneTextSelectionColor	Black
PhoneAccentColor	#FF1BA1E2

Tabla 2. Colores del sistema.

Clave	Color
PhoneAccentBrush	PhoneAccentColor
PhoneInactiveBrush	PhoneInactiveColor
PhoneTextBoxBrush	PhoneTextBoxColor
PhoneBackgroundBrush	PhoneBackgroundColor
PhoneDisabledBrush	PhoneDisabledColor
PhoneBorderBrush	PhoneBorderColor
PhoneContrastForegroundBrush	PhoneContrastForegroundColor
PhoneSubtleBrush	PhoneSubtleColor
PhoneTextSeleccctionBrush	PhoneTextSelectionColor
PhoneContrastBackgroundBrush	PhoneContratsBackgroundColor
PhoneForegroundBrush	PhoneForegroundColor
TransparentBrush	Transparent

Tabla 3. Estilos de brocha del sistema

Clave	Fuente
PhoneFontFamilyNormal	Segoe WP
PhoneFontFamilySemiLight	Segoe WP Light
PhoneFontFamilySemiBold	Segoe WP Semilight
PhoneFontFamilyLight	Segoe WP Semibold

Tabla 4. Fuentes del sistema

Clave	Tamaño
PhoneFontSizeSmall	14pt
PhoneFontSizeNormal	15pt
PhoneFontSizeMedium	17pt
PhoneFontSizeMediumLarge	19pt
PhoneFontSizeLarge	24pt
PhoneFontSizeExtraLarge	32pt
PhoneFontSizeExtraExtraLarge	54pt

Tabla 5. Tamaños de fuente del sistema

Una causa bastante común de fallo en la aceptación de una aplicación en el Marketplace suele ser tener textos en color blanco. Si el usuario cambia el tema de fondo a blanco, los textos no se verían. Para evitar

esto hemos optado por establecer a mano el fondo de todas las áreas en la que aparezcan textos, dejando las letras en blanco. Por lo que de cara a la legibilidad de nuestros textos es intrascendente que el tema del fondo del sistema sea blanco o negro.

d) Estilos

Con los estilos podemos establecer el color, tamaño, y tipo de letra de todos los textos de la aplicación. Un estilo es un recurso que permite agrupar y establecer el valor de las propiedades de un tipo de elemento que deseemos para poder aplicarlas más tarde, ya sea explícitamente mediante el uso de un nombre de recurso, o implícitamente mediante la especificación del tipo de elemento al cual va dirigido el estilo.

Por ejemplo, podemos definir un estilo para un *ListBox* al cual queremos cambiarle el color, tamaño y fuente de la letra. En vez de especificar directamente estos valores en todos los *ListBox* que lo necesitemos, vamos a hacerlo en un recurso del archivo app.xaml:

```
<Style x:Key="estiloListBox" TargetType="ListBox">
    <Setter Property="Foreground" Value="White"></Setter>
    <Setter Property="FontSize" Value="38"></Setter>
    <Setter Property="FontFamily" Value="Comic Sans MS"></Setter>
</Style>
```

Como podemos ver, dentro de un objeto *Style* asignamos *Setters* que son los encargados de establecer el valor de la propiedad indicada. También indicamos la propiedad *TargetType* al objeto *Style* para que resuelva en tiempo de diseño el tipo de elementos al que va destinado, de esta forma en la propiedad *Property* de los *Setter* obtendremos solo las propiedades que son válidas para el tipo indicado.

Como si de cualquier otro tipo de recurso se tratase, usamos una expresión *StaticResource*, para indicarle a la propiedad *Style* de los *ListBox* de la aplicación, que usen el estilo que hemos creado:

```
<ListBox Name="listaDias" Style="{StaticResource estiloListBox}"
    SelectedItem="{Binding ItemSeleccionado}"
    SelectionChanged="lstDias_SelectionChanged"
    ItemsSource="{Binding ListaDias}" >
</ListBox>
```

e) Plantillas.

Una de las grandes ventajas que aporta Silverlight es la capacidad de separar la apariencia de un elemento de su funcionalidad. De esta forma un botón tiene por su lado la apariencia y por el otro, totalmente aislada, está su funcionalidad de poder ser pulsado, lanzar eventos, etc. Esto permite algo muy útil, el poder sobrescribir la apariencia totalmente sin afectar a su funcionalidad.

Como vimos anteriormente, la clase base *Control* añade a los elementos una propiedad llamada *Template* del tipo *ControlTemplate*. En esta clase es donde podemos especificar la plantilla a usar para visualizar nuestro control, es decir, los elementos que componen su apariencia.

Podemos crear y aplicar una plantilla a los elementos mediante un estilo. Como vimos anteriormente, un estilo establece el valor por defecto de las propiedades que especifiquemos de un elemento y entre ellas podemos especificar el valor de la propiedad *Template* directamente:

```
<Style x:Key="estiloListBoxItem" TargetType="ListBoxItem">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="ListBoxItem">
                <Border x:Name="LayoutRoot">
                    <VisualStateManager.VisualStateGroups>
                        <!--Gupos visuales Seleccionado o Deseleccionado-->
```



```

<VisualStateGroup x:Name="SelectionStates">
<VisualState x:Name="Unselected"/>
<VisualState x:Name="Selected">
<Storyboard>
<!--Jugamos con el color-->
<ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground"
Storyboard.TargetName="ContentContainer">
<DiscreteObjectKeyFrame KeyTime="0" Value="White"/>
<DiscreteObjectKeyFrame KeyTime="0:0:1">
<DiscreteObjectKeyFrame.Value>
<!--Color que se queda al final-->
<SolidColorBrush Color="Red"/>
</DiscreteObjectKeyFrame.Value>
</DiscreteObjectKeyFrame>
</ObjectAnimationUsingKeyFrames>
<!--Jugamos con el movimiento-->
<DoubleAnimationUsingKeyFrames
Storyboard.TargetProperty="(UIElement.Projection).(PlaneProjection.RotationX)"
Storyboard.TargetName="ContentContainer">
<EasingDoubleKeyFrame KeyTime="0" Value="0"/>
<EasingDoubleKeyFrame KeyTime="0:0:1" Value="360"/>
</DoubleAnimationUsingKeyFrames>
</Storyboard>
</VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
<ContentControl x:Name="ContentContainer"
ContentTemplate="{TemplateBinding ContentTemplate}"
Content="{TemplateBinding Content}"
Foreground="{TemplateBinding Foreground}">
</ContentControl>
</Border>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>

```

En el código anterior, el recurso *ControlTemplate* está especificado dentro del valor de un *Setter* del estilo y es este último el que establece la propiedad *x:key*. De esta forma lo aplicaríamos al elemento, como si de un estilo normal se tratase:

```

<ListBox Name="listaViajes" Style="{StaticResource estiloListBox}"
DisplayMemberPath="Nombre"
ItemContainerStyle="{StaticResource estiloListBoxItem}"
SelectedItem="{Binding ItemSeleccionado}"
ItemsSource="{Binding Lista}">
</ListBox>

```

El estilo anterior está asignado a un *ListBox* en el cual hemos querido añadir una animación para evitar que la aplicación sea muy monótona. De tal forma que cuando el usuario selecciona un elemento de la lista el elemento da una vuelta sobre sí mismo para terminar cambiando de color.

Todo esto lo definimos en la plantilla usando la clase *VisualStateManager*, que gestiona los cambios de aspecto de nuestra plantilla cuando cambiemos el estado del elemento. Dentro del *VisualStateManager* podemos definir grupos visuales como *MouseOver*, *Normal*, *Pressed* o *Disabled*.

Una vez definido el estado sobre el que queremos trabajar podemos usar animaciones. En este caso la animación consiste en que las letras se voltean sobre sí mismas. Para definir las animaciones y el tiempo que queremos que tarde la transición de un estado a otro usamos la clase *Storyboard*.

IV. Página principal de nuestra aplicación: navegación, comandos

La primera pantalla de nuestra aplicación va a ser muy sencilla, solo tendremos dos botones que nos permitirán ir hacia la parte de planificaciones de viaje o hacia la parte que se utilizará mientras el usuario está de viaje. Por ello vamos a ver cómo navegar entre las distintas pantallas de la aplicación y también veremos el concepto Comando, que será utilizado constantemente.

- a. Interfaz: MainPage.xaml.
- b. Comandos.
- c. Vista modelo de la página: VMMainPage.cs.

a) Interfaz: MainPage.xaml

Podemos empezar a ver cómo trabajar con el patrón MVVM en la primera pantalla de nuestra aplicación, la cual consta simplemente de dos botones: “Planificar un Viaje” y “Durante el viaje” (Ilustración 19):



Ilustración 19. MainPage.xaml

La página XAML correspondiente a esta pantalla es muy sencilla, tenemos dos botones dentro de un *StackPanel* con los correspondientes enlaces a datos:

```
<StackPanel Grid.Row="1">
    <Button Content="Planificar un viaje" Height="100" Name="botónPlanificacion"
        Width="300" Command="{Binding IrPlanificacion}" Foreground="Gold"
        BorderBrush="Gold"/>
    <Button Content="Durante el viaje" Height="100" Name="botónViajes" Width="300"
        Command="{Binding IrViajes}" Foreground="Gold" BorderBrush="Gold"/>
</StackPanel>
```

b) Comandos

Lo interesante en esta interfaz sería ver el uso de los comandos. Con la propiedad de dependencia llamada *Command* de tipo *ICommand* nos encargamos de capturar el evento clic del botón, para lanzar la ejecución de nuestro comando, que nos permitirá navegar a la página *Planificacion.xaml*.

Los comandos se basan en la *interface ICommand*, localizada en el *namespace System.Windows.Input*, que nos permite llamar a un método de forma abstracta a través del método *Execute* de la *interface*. También podemos definir un método que evalúe con anterioridad si el comando puede ser ejecutado mediante el método *CanExecute*.

El uso de esta *interface ICommand* nos aporta dos grandes ventajas:

- Habilita que ejecutemos métodos definiéndolos como enlaces de datos en XAML, sin tener que escribir código extra ni usar eventos que ejecuten nuestra lógica en *code behind*.
- Al enlazar un elemento a un comando, si este dispone de un método que compruebe si puede ser ejecutado (*CanExecute*), nuestro elemento reaccionará a la respuesta de éste. Si no puede ser ejecutado se desactivará. Si puede ser ejecutado se activará, liberándonos de la necesidad de escribir código que habilite o deshabilite controles dependiendo de si puede o no ser ejecutado en un momento dado.

Una vez explicado qué es un comando vamos a ver cómo se usa. Debemos crear una clase que implemente el *interface ICommand*, implementar los métodos *Execute* y *CanExecute* y el evento *CanExecuteChanged*. Pero para evitar que cada comando tenga su propia clase y tener que declarar cada uno que deseemos usar como un recurso en nuestra página tenemos la posibilidad de crear un *DelegateCommand*. Así podremos exponer los comandos como propiedades de nuestra vista modelo y, de esta forma, unificar su acceso a través del *DataContext* de nuestra página.

DelegateCommand es una clase que implementa la infraestructura del *interface ICommand*, con el añadido de permitirnos usar un delegado para indicar qué método deseamos que ejecute el comando, y otro para el método que queremos que se encargue de comprobar si el comando puede ser ejecutado:

```
public class DelegateCommand : ICommand
{
    Action execute;
    Func<Boolean> CanExecuteMethod;
    public DelegateCommand(Action commandExecute, Func<Boolean> canExecuteMethod)
    {
        execute = commandExecute;
        CanExecuteMethod = canExecuteMethod;
    }

    bool ICommand.CanExecute(object parameter)
    {
        return CanExecuteMethod();
    }

    void ICommand.Execute(object parameter)
    {
        execute();
    }

    public event EventHandler CanExecuteChanged;

    public void RaiseCanExecuteChanged()
    {
        if (CanExecuteChanged != null)
            CanExecuteChanged(this, new EventArgs());
    }
}
```

Como se puede ver en el código, la clase *DelegateCommand* es una implementación de la *interface ICommand*, al que hemos añadido un constructor que admite como parámetro el método a ejecutar por el comando.

En la última parte de la clase tenemos el evento *CanExecuteChanged*, que debemos lanzar para notificar a la interfaz de usuario de que ha cambiado la posibilidad de ejecutar el comando. Por último tenemos el método *RaiseCanExecuteChanged* que deberemos ejecutar cuando queramos refrescar el estado de nuestro comando, por ejemplo, al cambiar una propiedad en nuestra vista modelo.

Esta clase la incluiremos dentro de la carpeta “Base”, que a su vez está dentro de la carpeta “VistaModelos”. No tendremos que volver a acceder a ella ya que es común para el resto de vistas modelos de nuestra aplicación.

c) Vista Modelo de la página: VMMainPage.cs

Lo siguiente que vamos a ver es la clase “VMMainPage” que actuará como nuestra vista modelo:

```
public class VMMainPage
{
    private ICommand irPlanificacion;
    private ICommand irViajes;
    public List<Viaje> listaViajes { get; set; }
```

Como vimos anteriormente en la página XAML, cuando se ejecuta el *binding* del primer botón estamos llamando al método “IrPlanificacion” a través del cual accedemos al método “irPlanificacionExecute” donde escribimos el código de la funcionalidad que queramos dar. En este caso estamos llamando a otra clase controladora de navegación, que posteriormente analizaremos. Este paso lo repetimos para darle funcionalidad al comando “IrViajes”. Será igual, salvo que nos dirigirá a otra pantalla en caso de que haya viajes creados:

```
public ICommand IrPlanificacion
{
    get
    {
        if (irPlanificacion == null)
            irPlanificacion = new DelegateCommand(irPlanificacionExecute,
            PuedeEjecutarIrPlanificacionExecute);

        return irPlanificacion;
    }
}
void irPlanificacionExecute()
{
    controlador.ControladorDeNavegacion.Current.NavigateTo("Planificacion");
}
```

En cuanto a la llamada a *PuedeEjecutarIrViajesExecute()* siempre devolverá el valor *true*, puesto que se trata simplemente de un botón que siempre estará operativo:

```
bool PuedeEjecutarIrPlanificacionExecute()
{
    return true;
}
```

Después de crear nuestra vista modelo tenemos que establecerla como la fuente de datos de nuestra vista:

```
public partial class MainPage : PhoneApplicationPage
{
    public MainPage()
    {
        InitializeComponent();
```

```

        this.DataContext = new viewmodels.VMMainPage();
    }
}

```

En todas nuestras vistas aparecerá el método “*InitializeComponent()*” cuyo cometido es cargar la página compilada de un componente.

V. Navegación en MVVM

En los anteriores apartados se explicaba cómo trabajar con el patrón MVVM. Concretamente en esta pantalla estamos siguiendo este patrón para poder navegar a la página *Planificacion.xaml* si pulsamos el botón “botónPlanificacion” o para navegar a la página *Viajes.xaml* si pulsamos el botón “botónViajes”. Para ello ejecutábamos esta línea de código:

```
controlador.ControladorDeNavegacion.Current.NavigateTo("Planificacion");
```

Como vemos, estamos llamando a un controlador de navegación. Que no es más que una clase que sigue el patrón *Singleton* (una entrada única a la clase, a través de una propiedad *Instance* que devuelve siempre la misma instancia de la clase). Esta clase se encarga de, a petición de la vista modelo, navegar hacia una página, con la posibilidad de enviar argumentos o ejecutar métodos en el proceso.

Esta clase llamada *NavigationController* expone su única instancia mediante la propiedad llamada *Current*. De esta forma, siempre usaremos nuestra clase a través de la propiedad *Current*, manteniendo una única instancia de la misma:

```

public class ControladorDeNavegacion
{
    static ControladorDeNavegacion current;
    public static ControladorDeNavegacion Current
    {
        get
        {
            if (current == null)
            {
                current = new ControladorDeNavegacion();
            }
            return current;
        }
    }
}

```

Lo siguiente que vemos en el código es una forma de identificar nuestras páginas para poder navegar a ellas. En este sentido lo más extendido es mantener un diccionario, con una clave como identificador y, como contenido, la Uri de la página a la que deseamos navegar:

```
static Dictionary<String, Uri> registeredViews = new Dictionary<String, Uri>();
```

A continuación creamos un constructor privado que se encargue de registrar las páginas en nuestro diccionario *registeredViews*. De esta forma, solo deberemos indicar el identificador al que deseamos navegar y el diccionario usará su Uri, siendo más limpio y teniendo un punto único donde cambiar la Uri si movemos las vistas:

```

private ControladorDeNavegacion()
{
    registeredViews.Add("Planificacion",
        new Uri("/Planificacion.xaml", UriKind.Relative));
    registeredViews.Add("Viajes",
        new Uri("/Viajes.xaml", UriKind.Relative));
}

```

Ahora vamos a profundizar en el método *NavigateTo(String destino)*. Es el que se encarga de la navegación, que en este caso es sin parámetros, simplemente vamos a una página. Más adelante veremos el paso de parámetros.

Un problema que nos encontramos en este paso es que el servicio “NavigationService” solo está disponible en las páginas, y no podemos acceder a él desde una clase. Para solucionarlo, hemos usado la propiedad *RootVisual* de *App.Current*, que contendrá el “Frame” de nuestra aplicación.

Al invocar al método le pasamos como parámetro la clave que tenemos registrada en nuestro diccionario interno y usamos “RootVisual” para obtener el “Frame” principal:

```
public void NavigateTo(String destino)
{
    PhoneApplicationFrame rootFrame = App.Current.RootVisual as
    PhoneApplicationFrame;
    rootFrame.Navigate(registeredViews[destino]);
}
```

Por último tenemos el evento que se lanza una vez que hemos navegado a la página destino:

```
void root_Navigated(object sender, NavigationEventArgs e)
{
    NavigationMethod(e);
}
```

Aparte de los métodos que nos permiten navegar hacia siguientes páginas, hemos de implementar también un método para retroceder.

VI. Navegación hacia atrás

Una de las especificaciones mínimas que Microsoft exige para Windows Phone es que el dispositivo tenga tres botones hardware: Botón inicio, botón buscar y botón atrás.

Este último botón cumple con dos funciones que deben tener un comportamiento exacto al esperado. De lo contrario nuestra aplicación será rechazada al certificarla en el Marketplace.

En primer lugar, el botón atrás cumple la función de terminar el uso de una aplicación y cerrarla. Esto se produce cuando nos encontramos en la primera página de la aplicación (en nuestro caso *MainPage.xaml*) y no queda historial hacia atrás que recorrer. Al presionar atrás volveremos al menú inicial del teléfono y nuestra aplicación se cerrará.

En segundo lugar, si hemos navegado entre páginas y tenemos un historial de páginas en las que hemos estado anteriormente, el botón atrás nos permitirá recorrer este historial. Cada vez que lo pulsemos abriremos la última página anterior a la actual que esté en el historial interno de navegación.

Aunque debemos conservar este comportamiento, sobre todo el primer caso pues es necesario para la certificación de la aplicación, podemos modificar un poco el segundo caso para dar fluidez a la navegación hacia atrás.

Por ejemplo, en la página *NuevoPlan.xaml*, tenemos un formulario con un botón de confirmación para guardar los cambios. Al presionar ese botón, en el código subyacente guardamos los datos que el usuario ha introducido y navegamos a la página desde la que accedimos al formulario. Si no modificásemos nada, la página *NuevoPlan.xaml* quedaría apilada en la pila de páginas del sistema y cuando quisiéramos salir de la aplicación habría que recorrerla entera, pasando otra vez por el formulario.

Lo que vamos a hacer es sacar de la pila la página que abandonamos para luego no tener que volver a pasar por ella. El método que vamos a utilizar para esta modificación es el *BackTo()*. En él simplemente

obtenemos el *Frame* y llamamos al método *GoBack()*, el cual navega al elemento más reciente del historial de navegación, es decir, navegamos hacia atrás en vez de llamar a la siguiente:

```
public void BackTo()
{
    PhoneApplicationFrame rootFrame = Application.Current.RootVisual as
    PhoneApplicationFrame;
    rootFrame.GoBack();
}
```

Un ejemplo de navegación para salir de la aplicación sin haber usado el método *BackTo()* sería:

MainPage.xaml > Planificacion.xaml > NuevoPlan.xaml > Planificacion.xaml (En este punto el usuario decide salir de la aplicación) > NuevoPlan.xaml > Planificacion.xaml > MainPage.xaml.

Mientras que si usamos el método *BackTo()* la navegación hasta salir sería:

MainPage.xaml > Planificacion.xaml > NuevoPlan.xaml > Planificacion.xaml (En este punto el usuario decide salir de la aplicación) > MainPage.xaml.

Con esta clase *NavigationController* hemos podido realizar la navegación, de una forma clara y limpia desde nuestra vista modelo.

VII. La base de datos

Antes de continuar viendo distintas pantallas de la aplicación vamos a ver cómo trabajar con bases de datos en Windows Phone y las clases que vamos a necesitar.

Lo primero de todo es hacer el diseño de las tablas que necesitaremos. La principal será la tabla Viajes que consta de seis columnas cuyos campos son (Tabla 6):

- **ViajeID:** Es el identificador de cada viaje, por lo tanto es único para cada uno de ellos y será la clave primaria.
- **Nombre:** Este campo guarda el nombre del viaje.
- **FechaIni:** Este campo se corresponde con la fecha del inicio del viaje.
- **FechaFin:** Este campo se corresponde con la fecha del fin del viaje.
- **Duracion:** En este campo guardamos la duración en días del viaje, este campo lo rellenara automáticamente la aplicación.
- **Presupuesto:** En este campo se puede guardar un presupuesto para realizar el viaje introducido por el usuario.

ViajeID	Nombre	FechaIni	FechaFin	Duracion	Presupuesto
1	Fjordos Noruegos	15/06/2013	30/06/2013	15	2700
2	Malaga	02/11/2012	04/11/2012	2	350
3	Lisboa	09/02/2013	13/02/2013	4	500

Tabla 6. Ejemplo de tabla Viajes

Complementando la tabla Viajes vamos a tener la tabla Dias, en la que cada entrada se corresponderá con un día de viaje, de tal forma que constará de los siguientes campos (Tabla 7):

- **DiaId:** Es el identificador único de cada día, será la clave primaria de esta tabla.

- **ViajeId:** A través de este campo tendremos una relación de clave foránea con la tabla Viajes, de tal forma que los días que tengan el mismo identificador de viaje corresponderán a un mismo viaje.
- **NumDia:** Se trata de la posición que ocupa el día con respecto al total del viaje, es decir, si es el primero, el cuarto, o el quinto.
- **Descripcion:** El usuario podrá introducir texto con la información de lo que quiera hacer ese día de viaje, la cual se guarda en este campo.

DiaId	ViajeId	NumDia	Descripcion
1	2	1	Llegada al hotel
2	2	2	Día en la playa
3	3	1	Visita familiares
4	3	2	Crucero por el rio
5	3	3	...
6	3	4	Vuelta a casa

Tabla 7. Ejemplo de tabla Dias

Por último tendremos la tabla Gastos, gracias a la cual el usuario podrá ir apuntando en todo momento en qué se ha gastado el dinero. Esta tabla tendrá los siguientes campos (Tabla 8):

- **GastoId:** Es el identificador único de cada gasto, será la clave primaria de esta tabla.
- **ViajeId:** A través de este campo tendremos una relación de clave foránea con la tabla Viajes, de tal forma que los gastos que tengan el mismo identificador de viaje corresponderán a un mismo viaje.
- **DiaId:** A través de este campo tendremos una relación de clave foránea con la tabla Dias, de tal forma que los gastos que tengan el mismo identificador de día se corresponderán a un mismo día.
- **Importe:** En este campo se guardará la cifra indicada por el usuario del importe del gasto.
- **Concepto:** En este campo se indicará en qué se ha producido el gasto.

GastoId	ViajeId	DiaId	Importe	Concepto
1	2	1	15	Propinas
2	2	1	32	Cena
3	2	2	5	Alquiler hamacas
4	3	4	60	Gasolina

Tabla 8. Ejemplo de tabla Gastos

Con Windows Phone 7.5 Microsoft ha introducido la posibilidad de usar bases de datos locales en nuestras aplicaciones mediante *SQL Server CE (Compact Edition)*.

SQL Server CE usa archivos .sdf (la extensión viene de las palabras inglesas *Server Database File*) para almacenar nuestra base de datos. En Windows Phone 7.5, estos archivos se guardan en nuestro almacenamiento aislado. En este sistema operativo una aplicación solo puede acceder al almacenamiento

aislado que le corresponde. Por este motivo, no podemos compartir una misma base de datos con varias aplicaciones.

Para crear la base de datos hemos seguido el modelo *code first* (código primero). En este modelo, hemos creado nuestras clases e indicado el mapeo a tablas y columnas de cada clase y propiedad.

A partir de estas clases se crea la base de datos en tiempo de ejecución en el dispositivo, aunque también existen herramientas que permiten crear clases a partir de una base de datos *sdf* que tengamos en tiempo de desarrollo. En esta versión de Windows Phone se implementa *LINQ TO SQL*² como *ORM* (mapeo objeto-relacional).

Para trabajar con *SQL Server CE* lo primero que necesitamos es añadir una referencia al ensamblado *System.Data.Linq* en nuestra aplicación. Para ello vamos a la ventana “Explorador de soluciones” de nuestro proyecto, hacemos clic con el botón derecho sobre la carpeta “References” y seleccionamos la opción “Agregar referencia” (Ilustración 20):

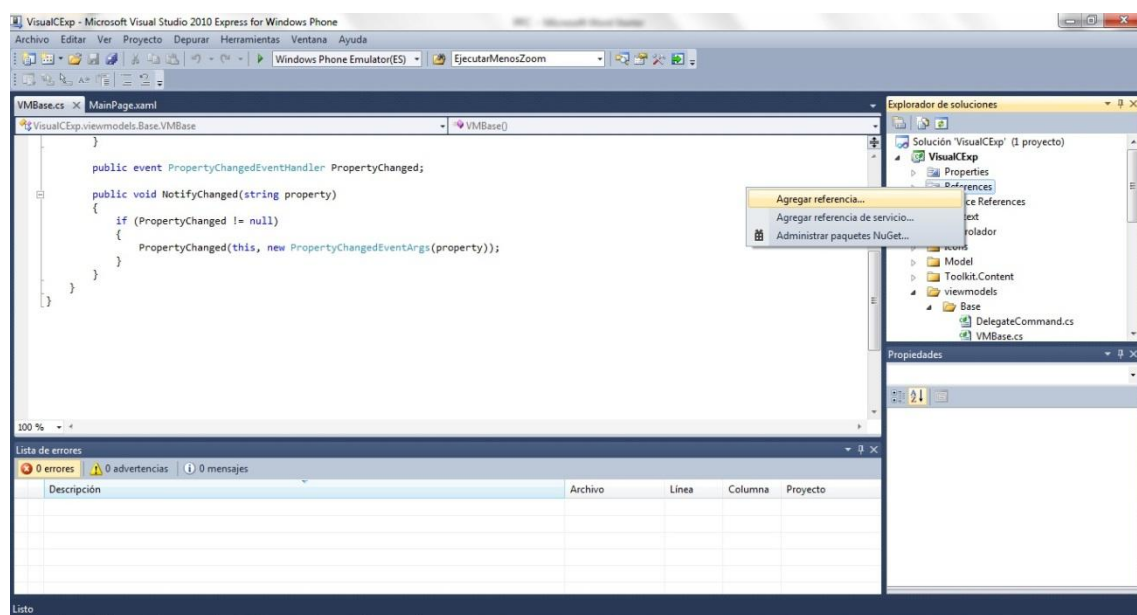


Ilustración 20. Agregar referencia al ensamblado.

Una vez que tenemos abierta la ventana “Agregar referencia”, nos desplazamos a la pestaña *.NET* en la que seleccionaremos el componente *System.Data.Linq* (Ilustración 21).

² LINQ TO SQL es un componente de .NET Framework 3.5 que proporciona una infraestructura en tiempo de ejecución para administrar los datos relacionales como objetos.

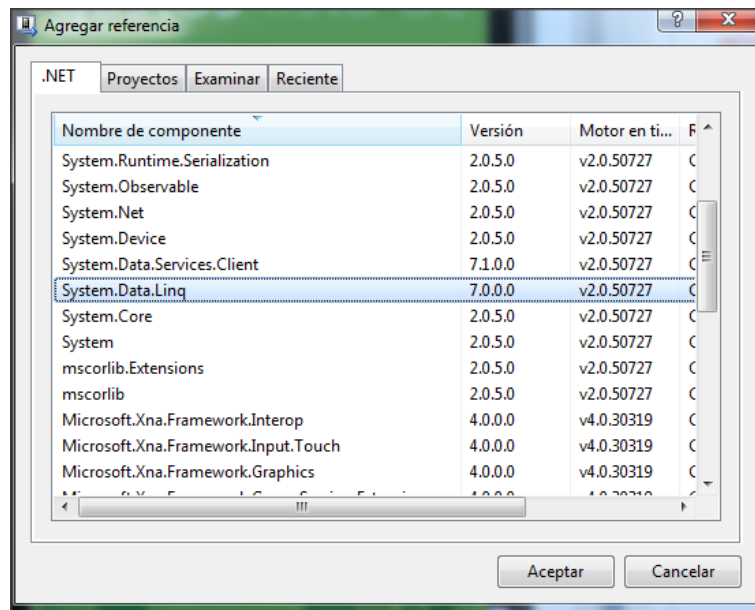


Ilustración 21. Agregar referencia al ensamblado.

También añadiremos un *using* del *namespace System.Data.Linq.Mapping* en nuestra clase Viaje. La cual quedará ubicada en el archivo Viaje.cs, cuya cabecera será:

```
using System;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Data.Linq.Mapping;
using System.Data.Linq;
using System.ComponentModel;
using AgendaViajesUc3m.VistaModelos.Base;

namespace AgendaViajesUc3m.Tablas
{
    [Table(Name = "Viajes")]
    public class Viaje : INotifyPropertyChanging
    {
```

En ella podemos almacenar la información de nuestros Viajes y añadir los atributos necesarios para poder mapearla a una tabla Viajes de nuestra base de datos.

La clase Viaje es una clase normal y corriente a la que hemos añadido atributos de *System.Data.Linq.Mapping*:

- **Table:** Atributo para la clase. Así la identificamos como una tabla, con el parámetro *Name* que nos permite especificar el nombre de la tabla relacionada en la base de datos (Viajes).

```
[Table(Name = "Viajes")]
public class Viaje : INotifyPropertyChanging
{
```

```

private int viajeId;
private String nombre;
private DateTime fechaIni;
private DateTime fechaFin;
private int duracion;
private String presupuesto;

```

- **Column:** Atributo para las propiedades. Indica si se trata de una clave primaria, si es autogenerada por la base de datos o si puede contener valores nulos.

```

[Column(IsPrimaryKey = true, IsDbGenerated = true)]
public int ViajeId
{
    get
    {
        return viajeId;
    }
    set
    {
        viajeId = value;
        this.OnPropertyChanging("ViajeId");
        NotificarCambio("ViajeId");
    }
}

[Column(CanBeNull = false)]
public String Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        nombre = value;
        this.OnPropertyChanging("Nombre");
        NotificarCambio("Nombre");
    }
}

```

Los ficheros correspondientes a las clases Viaje, Dia y Gasto, los agruparemos bajo una carpeta llamada Tablas, de cara a una mejor organización del proyecto.

Posteriormente se crea el contexto de datos de nuestra aplicación, llamado *AppDbContext*. Este contexto heredará de la clase *DataContext* del ensamblado *System.Data.Linq*, y expondrá toda la funcionalidad necesaria para poder trabajar con nuestra base de datos.

Se trata de una clase muy sencilla, que ubicaremos dentro de la carpeta Contexto. Al heredar de *DataContext* ya tenemos toda la funcionalidad requerida implementada en nuestra clase, simplemente añadimos una propiedad de tipo “*Table*” de Viaje, de Dia y otra de Gasto para exponer nuestras tablas Viajes, Dias y Gastos.

El constructor de la clase *AppDbContext* requiere un parámetro que contenga la cadena de conexión hacia la base de datos que deseamos usar:

```

public class AppDbContext : DataContext
{
    public AppDbContext(string cadenaDeConexion)
        : base(cadenaDeConexion) { }

    public Table<Viaje> Viajes
    {

```

```

        get { return this.GetTable<Viaje>();}
    }
    public Table<Dia> Dias
    {
        get { return this.GetTable<Dia>();}
    }
    public Table<Gasto> Gastos
    {
        get { return this.GetTable<Gasto>();}
    }
}

```

Para realizar nuestra aplicación, no solo tenemos la clase Viaje correspondiente a la tabla Viajes de nuestra base de datos, sino que también trabajamos con las clases Dia y Gasto, a las que les corresponden las respectivas tablas en la base de datos llamadas Dias y Gastos. De tal modo que hay que establecer una relación de clave foránea entre las dos tablas.

Con el atributo *Association* indicamos a nuestro contexto que se trata de una propiedad de asociación. El parámetro *Storage* indica la columna a usar para almacenar la relación, y el parámetro *OtherKey* indica el campo de la clase Dia a usar para realizar la relación:

```

[Association(Storage = "Dias", OtherKey = "ViajeId")]
public EntitySet<Dia> Dias { get; set; }

[Association(Storage = "Gastos", OtherKey = "ViajeId")]
public EntitySet<Gasto> Gastos { get; set; }

```

Como estamos trabajando con dispositivos móviles, hemos tenido en cuenta que su hardware tiene ciertas limitaciones y por tanto siempre hay que intentar optimizar el rendimiento de la aplicación. Para poder optimizar el uso de bases de datos locales hemos usado la optimización de *INotifyPropertyChanging*.

El seguimiento de cambios en *LINQ TO SQL* funciona manteniendo dos copias de cada objeto recuperado. Una copia se mantiene tal y como ha sido recuperada desde la base de datos, y la otra es la que se podrá modificar en la aplicación. De esta forma, cuando analizamos una entidad, *LINQ TO SQL* puede saber qué propiedades han cambiado y actualizarlas.

Esto se realiza para todos los elementos que recuperamos, sean modificados o no, por lo que en este último caso no existe razón alguna para mantener dos copias de cada elemento en memoria.

Usando el *interface INotifyPropertyChanging*, del *namespace System.ComponentModel*, podemos notificar a nuestro contexto cuándo se está modificando un elemento, para que obtenga la copia original de ese elemento en concreto, ya que así indicamos a *LINQ TO SQL* que valores han cambiado. Sólo los elementos que sabemos que han cambiado de valor, serán modificados en memoria. Es decir, en vez de mantener dos copias de la entidad y luego compararlas para ver los cambios producidos, directamente le indicamos que se han producido cambios y por tanto los hacemos directamente en la copia original.

En nuestra clase Viaje, además de implementar el *interface INotifyPropertyChanging*, hemos añadido un método al que llamaremos desde el *set* de las propiedades y que lanzará el evento *PropertyChanging*:

```

public event PropertyChangedEventHandler PropertyChanging;

public void OnPropertyChanging(string propertyName)
{
    if (PropertyChanging != null)
        PropertyChanging.Invoke(this, new PropertyChangedEventArgs(propertyName));
}

```

VIII. Página donde gestionar las planificaciones de viaje: Barra de aplicación, ListBox, enlace a datos

En este apartado vamos a ver nuevos elementos a la par que navegamos hacia la página `Planificacion.xaml`. En ella introducimos uno de los elementos más utilizados en Windows Phone 7.5, el `ListBox`, que nos permitirá mostrar listas de cualquier tipo de elemento y explicaremos en profundidad el uso del enlace a datos:

- Página `Planificacion.xaml`
- Barra de aplicación.
- Métodos de navegación en *code behind*.
- Vista modelo de la página `Planificacion.xaml`.
- La propiedad *Mode*.
- Sistema de notificación de cambios.

a) Página `Planificacion.xaml`

Para navegar a esta página el usuario ha tenido que pulsar el botón “Planificar un viaje”, la funcionalidad del otro botón la analizaremos más adelante (Ilustración 22):



Ilustración 22. `MainPage.xaml` y `Planificacion.xaml`

Nos situamos en la página de la derecha, en la que tenemos el título y subtítulo de la página, una lista vacía en la que se irán mostrando las planificaciones de viajes y una barra en el borde inferior con tres botones. En esta misma página se irán añadiendo a la lista las distintas planificaciones de viaje que vaya creando el usuario. (Ilustración 23):



Ilustración 23. Planificacion.xaml

Como vemos en la imagen, los botones correspondientes a editar y eliminar siguen apareciendo deshabilitados. Mientras que el usuario no seleccione una planificación, estos botones no se habilitarán (Ilustración 24):

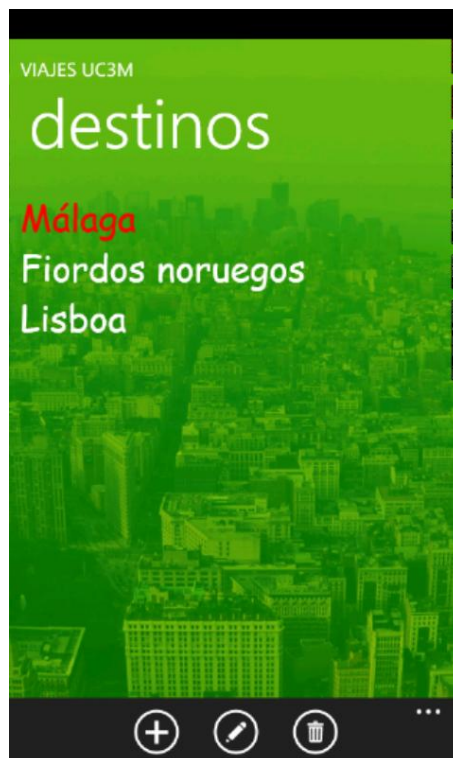


Ilustración 24. Botones editar y eliminar habilitados.

b) Barra de aplicación

La barra inferior que se ve en las imágenes es uno de los elementos de la interfaz de usuario que tenemos a nuestra disposición. Se trata del *AppBar* de Windows Phone, el cual se compone de dos partes bien diferenciadas (Ilustración 25):

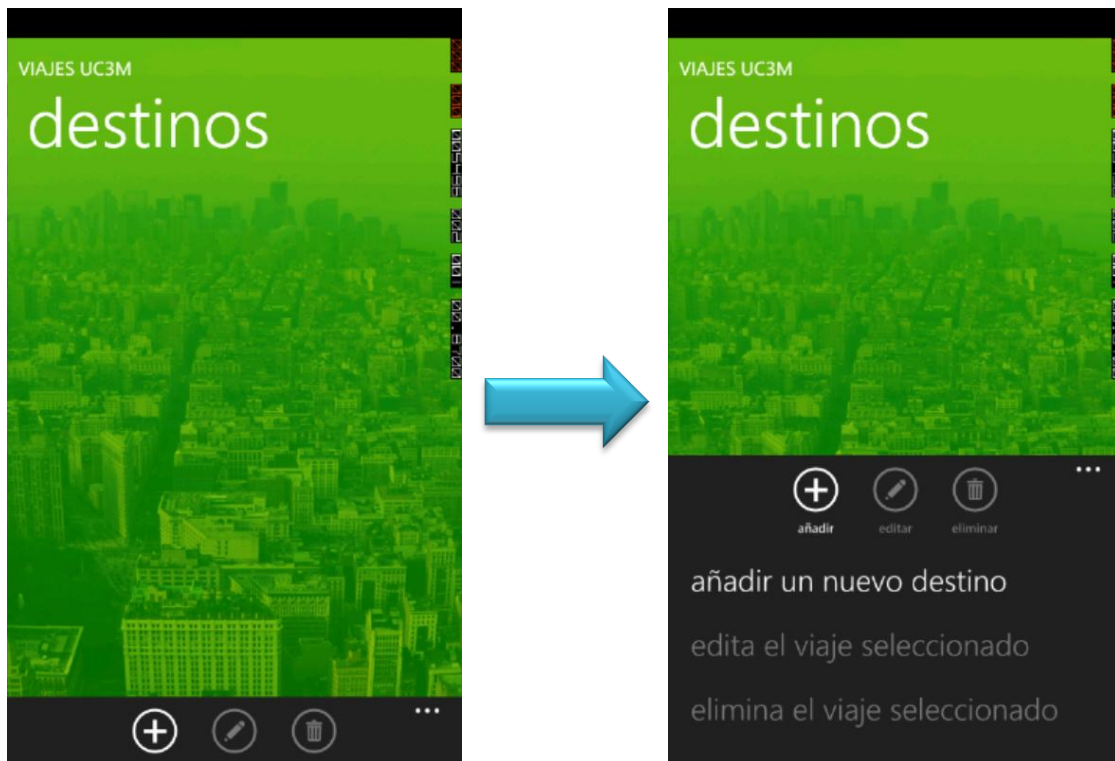


Ilustración 25. Botones y menú del elemento *AppBar*.

En primer lugar tenemos una línea de botones con iconos y un pequeño texto descriptivo, acompañados de tres puntos en la parte superior derecha. Esta línea de botones es lo único visible de la *AppBar* por defecto y los botones son del tipo *AppBarIconButton*.

Todos los iconos tienen un tamaño de 48x48 píxeles en total y son lo más significativos posibles. Estos iconos los hemos obtenido junto con el SDK de Windows Phone, ya que están incluidos en un kit de iconos listos para ser usados, aunque también se podrían haber creado iconos nuevos. Esos iconos los tenemos en la siguiente ruta:

- C:\ProgramFiles(x86)\MicrosoftSDKs\WindowsPhone\v7.1\Icons\dark

En segundo lugar disponemos de una lista de opciones, que solo pueden mostrar texto y están ocultas por defecto, mostrándose al presionar sobre los tres puntos que encontramos en la parte superior derecha de la barra. Estos menús pertenecen a la colección *MenuItems* y son del tipo *AppBarMenuItem*:

```
<phone:PhoneApplicationPage.AppBar>
  <shell:AppBar IsVisible="True" IsMenuEnabled="True">
    <shell:AppBarIconButton IconUri="\Icons\appbar.add.rest.png"
      Text="Añadir" Click="NuevaPlanificacion"/>
    <shell:AppBarIconButton IconUri="\Icons\appbar.edit.rest.png"
      Text="Editar"/>
    <shell:AppBarIconButton IconUri="\Icons\appbar.delete.rest.png"
      Text="Eliminar"/>
    <shell:AppBar.MenuItems>
      <shell:AppBarMenuItem Text="Añadir un nuevo destino"
        Click="NuevaPlanificacion"/>
      <shell:AppBarMenuItem Text="Edita el viaje seleccionado"/>
    </shell:AppBar.MenuItems>
  </shell:AppBar>
</phone:PhoneApplicationPage.AppBar>
```

```

        <shell:ApplicationBarItem Text="Elimina el viaje seleccionado"/>
    </shell:ApplicationBar.MenuItems>
</shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>

```

Al trabajar con este elemento debemos tener en cuenta que no basta con indicar la ruta en la que podemos encontrar los iconos de nuestra aplicación. Además, tenemos que configurar sus propiedades “Acción de compilación” y “Copiar en el directorio”.

La propiedad “Acción de compilación” controla el comportamiento del compilador respecto al archivo y marcamos la opción “Contenido” para que funcione correctamente según [4].

La otra propiedad que deberemos cambiar es “Copiar en el directorio”. La configuraremos como “Copiar si es posterior”, ya que el archivo se copia del directorio de proyecto al directorio “bin” la primera vez que se genera el proyecto. Cada vez que se vuelve a generar el proyecto, se compara la propiedad “Fecha de modificación” de los archivos. Si el archivo en la carpeta de proyecto es posterior, se copia en la carpeta “bin”, reemplazando el archivo que se encuentra en ella. Si el archivo de la carpeta “bin” es posterior, no se copia ningún archivo. Esta configuración almacena los cambios realizados en los datos en tiempo de ejecución, por lo que cada vez que ejecuta la aplicación y guarda cambios en los datos, estos cambios son visibles la próxima vez que ejecuta la aplicación (Ilustración 26):

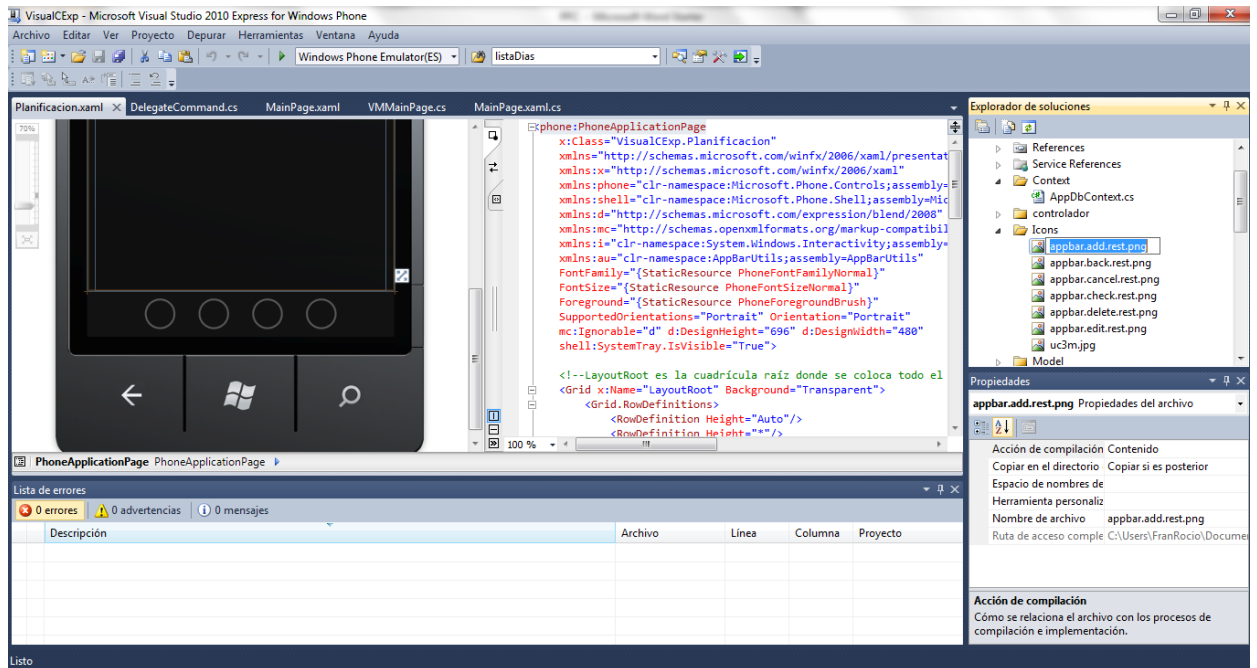


Ilustración 26. Propiedades del elemento *ApplicationBarIconButton*.

Otra cosa a tener en cuenta al trabajar con *ApplicationBar* es que ni *ApplicationBarIconButton* ni *ApplicationBarMenuItem* son objetos de Silverlight, por lo que no soportan enlace a datos ni propiedades adjuntas o de dependencia. Esto nos crea un problema a la hora de integrar estos objetos con el patrón MVVM. Para solventarlo hemos usado dos opciones:

Para el botón “Añadir” hemos incluido la llamada al manejador de eventos “click”:

```

<shell:ApplicationBarIconButton IconUri="\Icons\AppBar.Add.Rest.png" Text="Añadir"
Click="NuevaPlanificacion"/>

```


De tal forma que solo tenemos que ejecutar el comando correspondiente en el archivo de *code behind* Planificacion.xaml.cs:

```
private void NuevaPlanificacion(object sender, EventArgs e)
{
    contexto.IrNuevoPlan.Execute("{Binding IrNuevoPlan}");
}
```

En cambio para los botones “Editar” o “Eliminar” hemos utilizado el paquete *AppBarUtils* [5] que ya implementa funcionalidad (en el apartado 3.V.c) de la memoria se explica cómo añadir este tipo de paquetes) para poder enlazar comandos a nuestro *AppBar*, ya que este paquete contiene su propio *AppBar*, *AppBarIconButton* y *AppBarMenuItem*. Sólo tenemos que añadir las siguientes líneas de código al final del fichero Planificacion.xaml para añadir la funcionalidad del nuevo paquete:

```
<i:Interaction.Behaviors>
    <au:AppBarItemCommand Id="Eliminar" Command="{Binding ComandoEliminar,
Mode=TwoWay}"/>
    <au:AppBarItemCommand Type="MenuItem" Id="Elimina el viaje seleccionado"
Command="{Binding ComandoEliminar, Mode=TwoWay}"/>
    <au:AppBarItemCommand Id="Editar" Command="{Binding ComandoEditar,
Mode=TwoWay}"/>
    <au:AppBarItemCommand Type="MenuItem" Id="Edita el viaje seleccionado"
Command="{Binding ComandoEliminar, Mode=TwoWay}"/>
</i:Interaction.Behaviors>
```

Como vemos en el código XAML, creamos un *AppBarItemCommand* por cada botón o menú que queremos enlazar a datos, de tal forma que la propiedad *Id* debe contener el mismo valor que la propiedad *Text* del botón, y en caso de que estemos enlazando un menú, deberemos indicarlo estableciendo el valor *MenuItem* en la propiedad *Type*.

En el código también aparece configurada la propiedad *Mode*, más adelante, en el apartado “e)” de este capítulo explicaremos su uso.

c) Métodos de navegación en *code behind*

Hay que mencionar un método que hemos utilizado en el *code behind* de la página, es decir en el archivo Planificacion.xaml.cs, gracias al cual hemos conseguido dotar de gran fluidez a nuestra aplicación. Se trata del método *OnNavigatedTo()*. Este método es llamado cuando una página se convierte en la página activa de nuestra aplicación. Existe también el método complementario *OnNavigatedFrom()*, que es llamado cuando la página activa hasta ese momento deja de serlo.

Ambos métodos reciben un parámetro de tipo *NavigationEventArgs*. Con él podremos conocer si la navegación se está produciendo dentro de nuestra aplicación, el contenido de la página de destino a la que navegamos, el tipo de navegación o la Uri del destino de navegación.

Dentro del método *OnNavigatedTo()* vamos a comprobar el tipo de navegación mediante un *if*, de tal manera que si estamos volviendo a la página, volvamos a cargar la vista modelo en el contexto de la página:

```
base.OnNavigatedTo(e);
String modo = String.Format(e.NavigationMode.ToString());
if (modo.Equals("Back"))
{
    contexto = new VMPlanificacion();
    this.DataContext = contexto;
}
```

Con esto conseguimos refrescar la página con los valores actualizados. Si no volviésemos a cargar la vista modelo, al ser una navegación de tipo “Back”, la página se le mostraría al usuario tal y cómo la dejó antes de abandonarla.

d) Vista modelo de la página Planificacion.xaml

Para terminar de ver la funcionalidad de la pantalla de planificación vamos a analizar a continuación la clase correspondiente a su vista modelo, VMPlanificacion.cs.

En esta clase vamos a tener la lógica que se encarga de navegar y pasar los parámetros necesarios según se pulse un botón u otro. Además se incluirá el código necesario para eliminar elementos de la lista.

Lo primero que hacemos es editar el constructor por defecto. Para comenzar a trabajar con nuestra base de datos necesitamos un parámetro que contenga la cadena de conexión hacia la base de datos. En Windows Phone 7.5 el formato de cadena de conexión es especial. Como nuestra base de datos se encuentra en el almacenamiento aislado la cadena de conexión se compone de “Data Source=’isostore:/MiViajeDb.sdf’” y la ruta de almacenamiento aislado donde está nuestro archivo *sdf*.

A continuación hay que distinguir si es la primera vez que entramos en esta pantalla y por lo tanto tenemos que crear la base de datos, o por el contrario si la base de datos ya está creada, y simplemente queremos hacer un *Select* en ella para mostrar las planificaciones que han sido creadas anteriormente:

```
public VMPlanificacion()
{
    using (AppDbContext contextDb = new AppDbContext("Data
        Source='isostore:/MiViajeDb.sdf'"))
    {
        if (!contextDb.DatabaseExists())
        {
            contextDb.CreateDatabase();
        }
        else
        {
            Lista = (from Viaje viaje in contextDb.Viajes select viaje).ToList();
        }
    }
}
```

Para el caso de que la base de datos no exista tenemos que mostrar la pantalla al usuario vacía, es decir, sin ninguna lista, además de deshabilitar los botones de “editar” y “borrar”, ya que estos no se activarán hasta que el usuario haya seleccionado algún elemento de la lista.

Esto lo conseguimos implementando el método *PuedeEjecutarEditar()* de tal forma que si detecta que la lista está vacía, devuelve false y por tanto deshabilita el botón. El mismo código lo repetiríamos para el botón de eliminar:

```
public bool PuedeEjecutarEditar()
{
    bool dev = true;
    if (listaViajes == null)
    {
        dev = false;
    }
    else if (itemSeleccionado == null)
    {
        dev = false;
    }
    return dev;
}
```

Para el otro caso, en el que la base de datos ya está creada, sí que tenemos que mostrar una lista al usuario para que pueda seleccionar las distintas planificaciones que tenga creadas. Para ello usamos el elemento *ListBox*. Éste nos ofrece una propiedad llamada *ItemsSource*, en la que podremos establecer la expresión de enlace a datos que necesitemos para mostrar nuestra lista de elementos, en este caso al comando “Lista”.

El código XAML correspondiente a la lista sería el siguiente:

```
<ListBox Name="listaViajes" Style="{StaticResource estiloListBox}"
    DisplayMemberPath="Nombre"
    ItemContainerStyle="{StaticResource estiloListBoxItem}"
    SelectedItem="{Binding ItemSeleccionado, Mode=TwoWay}"
    ItemsSource="{Binding Lista, Mode=TwoWay}">
</ListBox>
```

Por defecto, el elemento *ListBox* intenta convertir cada elemento que encuentre en su propiedad *ItemSource* a una cadena para poder mostrar la información, lo que en este caso, llevaría a que mostrarse algo parecido a “*VisualCExp.Model.Viaje*”. Esta sería la representación en cadena del tipo de nuestros elementos. Para evitarlo tenemos la propiedad *DisplayMemberPath* donde podemos indicar el elemento *ListBox* cuyo nombre de la propiedad queremos que use para mostrar la información.

Otra propiedad con la que contamos y nos es de utilidad se llama *SelectedItem*, la cual combinada con el patrón MVVM nos permite obtener el elemento seleccionado para su posterior manipulación.

Por lo tanto, si en el constructor detecta que hay una lista creada, la interfaz tendría el siguiente aspecto (Ilustración 27):



Ilustración 27. Página con los distintos destinos ya creados.

Una vez que el usuario está en la pantalla de planificación, si selecciona un viaje y pulsa el botón de editar, se ejecuta el *binding* y llamamos al método *ComandoEditar*, a través del cual accedemos al método *EjecutarEditar()* donde escribimos el código de la funcionalidad que queramos dar, que en este

caso, será la de navegar a la página donde se podrá editar una planificación pasándole por parámetro el ID del viaje seleccionado:

```
public void EjecutarEditar()
{
    int viajeIdSeleccionado = (ItemSeleccionado as Viaje).ViajeId;
    string parameters = string.Format("ID={0}", viajeIdSeleccionado);
    controlador.ControladorDeNavegacion.Current.NavigateTo("EditaPlan", parameters);
}
```

Como *ItemSeleccionado* es un objeto de tipo *object*, tenemos que transformarlo en un objeto “Viaje”, para así poder acceder a su campo “ViajeID”. A continuación formamos la cadena *parameters* para incluirla en la navegación a la página *EditaPlan.xaml* como parámetro de navegación.

En cambio si el usuario selecciona un viaje y a continuación pulsa el botón de eliminar, se lanza el *binding* “ComandoEliminar”, el cual ejecuta el método *EjecutaEliminar()*. Lo primero que haríamos sería abrir conexión con la base de datos mediante el *using*:

```
public void EjecutarEliminar()
{
    using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
    {
```

A continuación obtenemos el identificador de viaje correspondiente al elemento seleccionado. Creamos una lista de la clase “Viaje” en la que guardar el viaje encontrado mediante su identificador, y otra lista de la clase “Dia” con los días encontrados con el mismo identificador de viaje:

```
int viajeIdSeleccionado = (itemSeleccionado as Viaje).ViajeId;
List<Viaje> listaAux = (from Viaje viaje in contextDb.Viajes where
viaje.ViajeId == viajeIdSeleccionado select viaje).ToList();
List<Dia> listaAux2 = (from Dia dia in contextDb.Dias where dia.ViajeId ==
viajeIdSeleccionado select dia).ToList();
List<Gasto> listaAuxGastos = (from Gasto gasto in contextDb.Gastos where
gasto.ViajeId == viajeIdSeleccionado select gasto).ToList();
```

Las siguientes líneas de código sirven para colocar ambas listas en un estado de eliminación pendiente de sus respectivas tablas y con la ejecución de la última línea se procede a dicha eliminación.

```
contextDb.Gastos.DeleteAllOnSubmit(listaAuxGastos);
contextDb.Dias.DeleteAllOnSubmit(listaAux2);
contextDb.Viajes.DeleteAllOnSubmit(listaAux);
contextDb.SubmitChanges();
```

Finalmente, se recupera de nuevo la lista con los viajes restantes y se notifica el cambio. Esto lo hacemos para refrescar la pantalla y reflejar así la eliminación del viaje, sin tener que recargar de nuevo la pantalla. Simplemente mediante las técnicas de enlace a datos y notificación de cambios reflejamos el cambio en la interfaz inmediatamente:

```
listaViajes = (from Viaje viaje in contextDb.Viajes select viaje).ToList();
NotifyChanged("Lista");
```

e) La propiedad *Mode*

Para poder terminar de entender el comportamiento de enlace a datos vamos a ver cómo funciona una propiedad muy importante que encontramos continuamente en el fichero XAML cuando utilizamos el enlace a datos. Ésta se refiere al modo del enlace, se llama *Mode* y puede tomar tres valores distintos: *OneWay*, *OneTime* y *TwoWay*:

- *OneWay* es el valor por defecto de todo enlace si no se indica lo contrario. Cuando se crea el enlace se obtiene el valor de la fuente de datos y se establece en la propiedad indicada. Si el valor cambia en la fuente de datos puede ser actualizado en la propiedad destino (pero no al contrario).
- *OneTime* indica al enlace que obtenga el valor de la fuente de datos cuando se establezca y nunca más, no teniendo la posibilidad de actualizar los cambios originados en la fuente de datos.
- *TwoWay* permite actualizar la fuente de datos con cambios desde el elemento enlazado, o el elemento enlazado con cambios en la fuente de datos.

f) Sistema de notificación de cambios.

Para que un enlace se actualice una vez que su valor ha cambiado, debemos implementar un sistema de notificación de cambios, usando la *interface INotifyPropertyChanged*.

Esta *interface* contiene un evento *PropertyChangedEventHandler* con el cual se puede notificar a nuestra interfaz de usuario de cambios en las propiedades enlazadas a datos. Usaremos este sistema para notificar los cambios en todas las pantallas de la aplicación.

Para implementar esta funcionalidad en la clase, hemos creado el evento que notificará los cambios y un método con el que podemos lanzar el evento, pasándole como parámetro nuestra instancia y el nombre de la propiedad que ha cambiado:

```
public event PropertyChangedEventHandler PropertyChanged;

public void NotificarCambio(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this,
            new PropertyChangedEventArgs(propertyName));
    }
}
```

Silverlight mantiene un registro por nombre de todas las propiedades enlazadas y este evento indicará que debe volver a evaluar el enlace de la propiedad indicada. Por último, en cada propiedad, en el *set* de cada una hay que llamar a nuestro método *NotificarCambio()*.

Para no tener que implementar esta *interface* en todas las vistas modelos de la aplicación nos creamos una clase llamada “VMBase” que se encontrará en la carpeta “Base” que a su vez está dentro de la carpeta “VistaModelos” de nuestro proyecto (Ilustración 28):

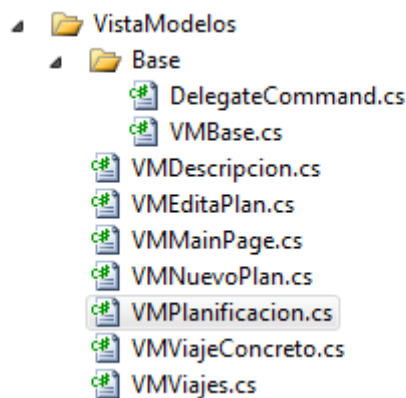


Ilustración 28. Subdirectorío de archivos pertenecientes a las vistas modelos.

De esta manera siempre que creamos una vista modelo añadiremos esta clase en su definición:

```
public class VMNuevoPlan : VMBase
{
```

Con esto se crea la infraestructura necesaria para que, cada vez que se modifique nuestra propiedad, la interfaz de usuario actualice su valor.

IX. Pantalla para crear una nueva planificación: DatePicker, inserciones en la BD, *tombstoning*

Continuamos viendo las pantallas de la aplicación. Recordamos que la primera vez que accedíamos a la misma, teníamos una lista vacía en la pantalla Planificacion.xaml. Para crear la primera planificación de viaje, el usuario pulsará el botón de nueva planificación y navegará hasta la página llamada NuevoPlan.xaml. En esta página veremos nuevas herramientas incluidas en el paquete *Toolkit*, la forma de hacer inserciones en la base de datos y por último veremos cómo conservar los datos de la aplicación en caso de que se produzca el *Tombstoning*:

- Página NuevoPlan.xaml.
- Herramienta para selección de fecha: *DatePicker*.
- Vista modelo de la página NuevoPlan.xaml.
- Método para calcular periodos de tiempo.
- Teclado numérico.
- Tombstoning*.

a) Página NuevoPlan.xaml

En esta página se nos presenta un sencillo formulario en el que el usuario debe introducir un nombre para la nueva planificación de viaje, una fecha de inicio y otra de regreso y por último un presupuesto máximo para la realización del mismo. Una vez que el formulario ha sido rellenado correctamente el usuario podrá pulsar el botón de “Aceptar” (Ilustración 29):



Ilustración 29. Formulario para crear planificaciones.

b) Herramienta para selección de fecha: *DatePicker*

Como vemos en esta pantalla, vamos a necesitar que el usuario seleccione una fecha. Para facilitarle la tarea vamos a incluir en nuestra aplicación un elemento similar al que se usa en el resto del sistema. El problema es que por defecto, hay ciertas utilidades que no tenemos disponibles, es el caso del *DatePicker*. Para estas ocasiones es cuando debemos instalar *frameworks* extras a través de NuGet como ya vimos en el capítulo 3.V.c.

Una vez instalado el *framework SilverlightToolkitWP* necesitamos añadir sus elementos a nuestra caja de herramientas.

Lo primero que tenemos que hacer es crear una nueva sección que se llame “Toolkit”. Para ello debemos pulsar con el botón derecho del ratón sobre la caja de herramientas y seleccionar la opción “Agregar ficha”. Se añadirá una nueva sección con el nombre en blanco para que escribamos el que deseemos (Ilustración 30):

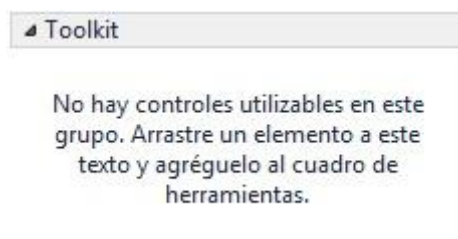


Ilustración 30. Nueva sección para los elementos del *Toolkit*.

A continuación, pulsamos de nuevo con el botón derecho sobre esta sección y seleccionamos la opción “Elegir elementos...”. Se desplegará la ventana de elementos de la caja de herramientas, ordenamos la lista por la columna Nombre del ensamblado y buscamos el ensamblado *Microsoft.Phone.Controls.Toolkit* (Ilustración 31):

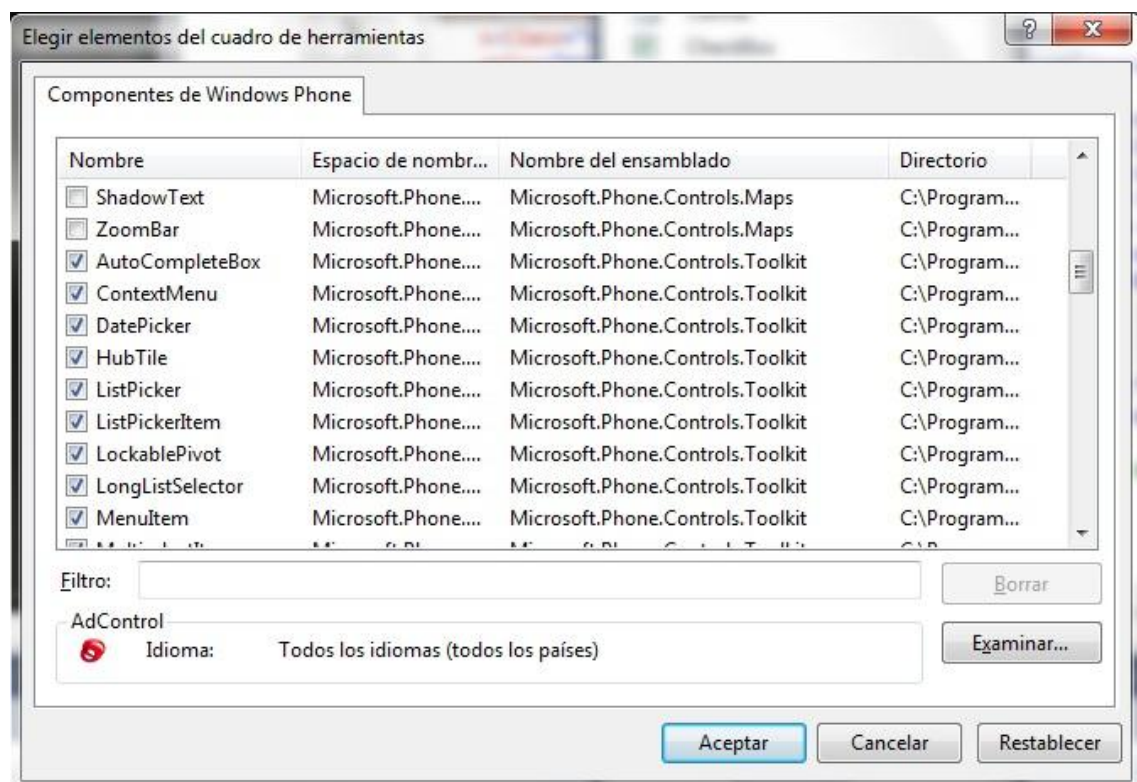


Ilustración 31. Elementos del *Toolkit*.

Simplemente debemos marcar los elementos del ensamblado *Microsoft.Phone.Controls.Toolkit* y presionar “OK” y ya tendremos todos los elementos del *Toolkit* en nuestra caja de herramientas (Ilustración 32):

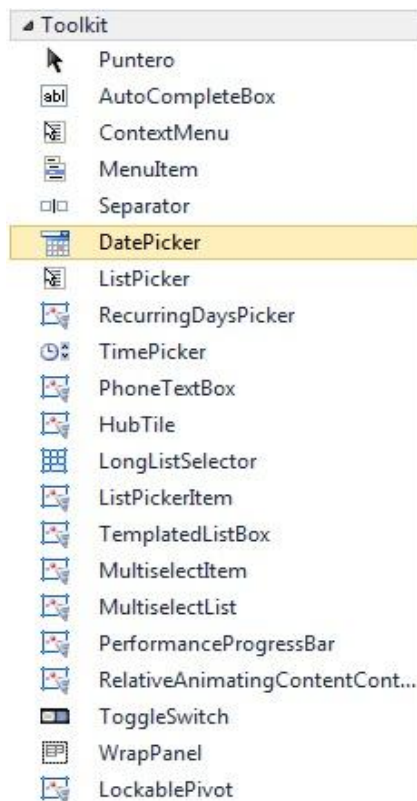


Ilustración 32. Caja de herramientas del *Toolkit*.

Una vez hecho esto solo tenemos que arrastrar el elemento que deseemos hasta nuestra página para poder usarlo. Al arrastrar y soltar un elemento del *Toolkit*, automáticamente nos creará la referencia al namespace en XAML:

```
xmlns:toolkit="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls.Toolkit"
```

Aun habiendo incluido el elemento *DatePicker* a nuestra página, aún faltan algunos detalles para que sea totalmente funcional, y es que el *DatePicker* que encontraríamos por defecto tendría el aspecto de la Ilustración 33.

Como se puede apreciar en la imagen, el formato de fecha no es el común que se usa en España, los textos del mes y día aparecen en inglés y los dos iconos de los botones son erróneos.

Vamos a empezar por arreglar los iconos de los botones. Para ello vamos a crear una carpeta en el directorio del proyecto que se llamará "Toolkit.Content". Hacemos clic con el botón derecho del ratón en el proyecto, seleccionamos la opción "Agregar" y después "Nueva carpeta". Escribimos el nombre de la carpeta "Toolkit.Content" (Ilustración 34).

Después tenemos que meter dos imágenes en esta carpeta llamadas: *ApplicationBar.Cancel.png* y *ApplicationBar.Check.png*. Las dos imágenes las podemos encontrar en la carpeta donde están los archivos binarios del Silverlight *Toolkit*, que son:

Para sistemas operativos de 64 bits:

- C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Toolkit\



Ilustración 33. Aspecto por defecto del elemento *DatePicker*.

Para sistemas operativos de 32 bits:

- C:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Toolkit\

Las imágenes están dentro de una subcarpeta llamada “Icons”, en la siguiente ruta:

- C:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Toolkit\Aug11\Bin\Icons

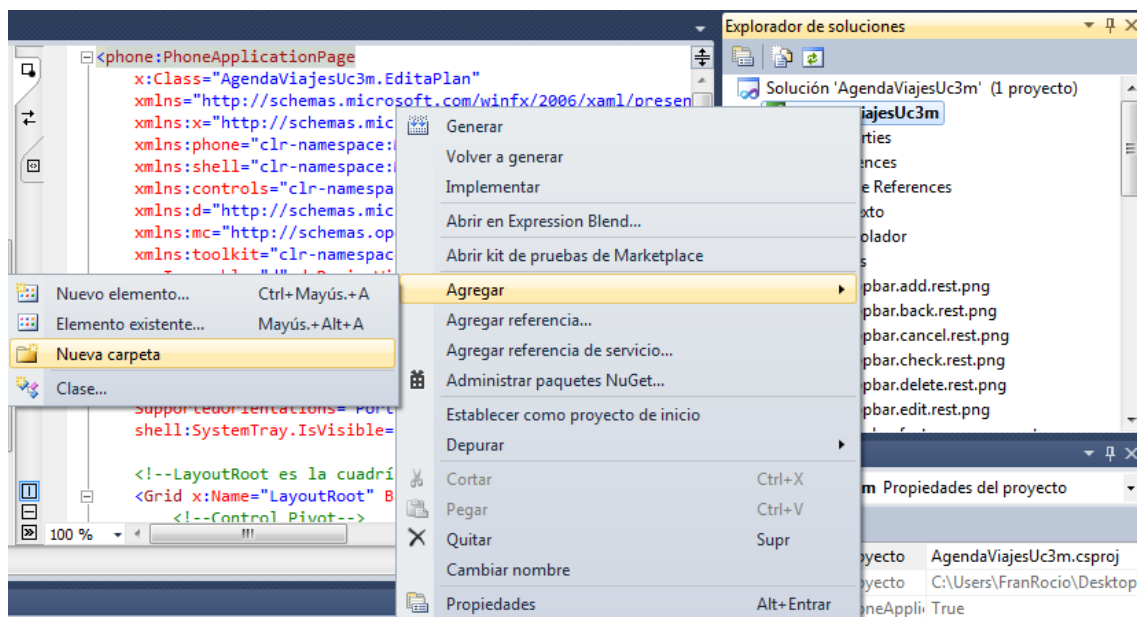


Ilustración 34. Creación de la carpeta *Toolkit.Content*.

Para agregar las imágenes, hacemos clic con el botón derecho del ratón sobre la carpeta Toolkit.Content y seleccionamos la opción "Agregar" y después "Elemento existente..." (Ilustración 35):

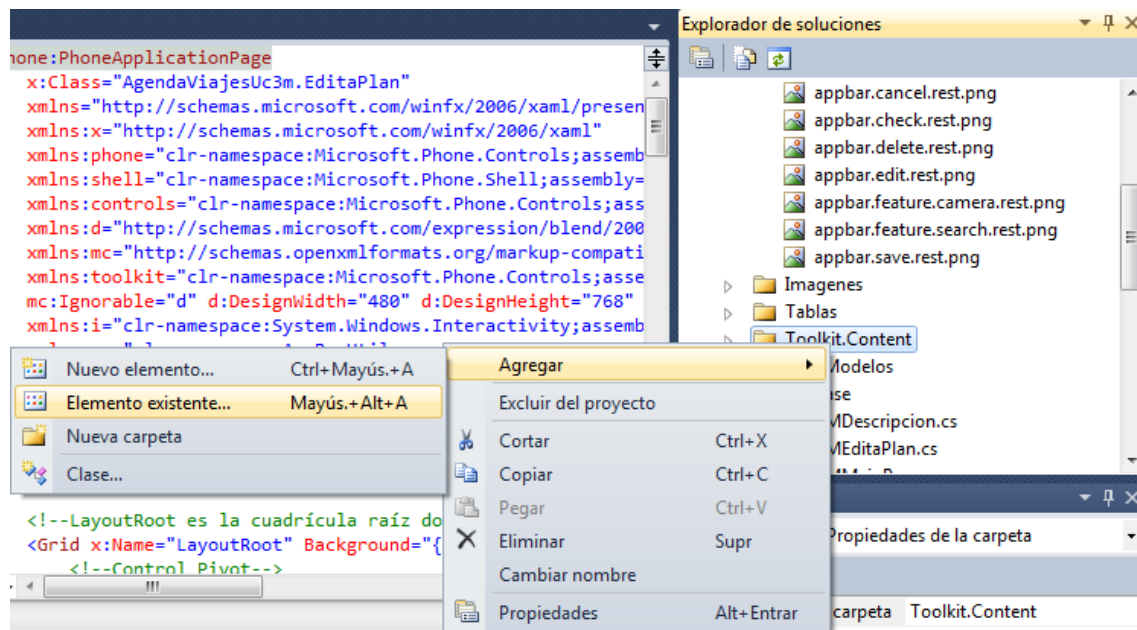


Ilustración 35. Agregar imágenes de los iconos.

Buscamos las dos imágenes y las agregamos. Después hay que modificar la propiedad “Acción de compilación” de cada imagen al valor “Contenido” y la propiedad “Copiar en el directorio” al valor “Copiar si es posterior”. Esta acción hay que repetirla para todos los iconos que queramos usar en nuestra aplicación, como ya comentamos en el capítulo 4.IX.b (Ilustración 36):

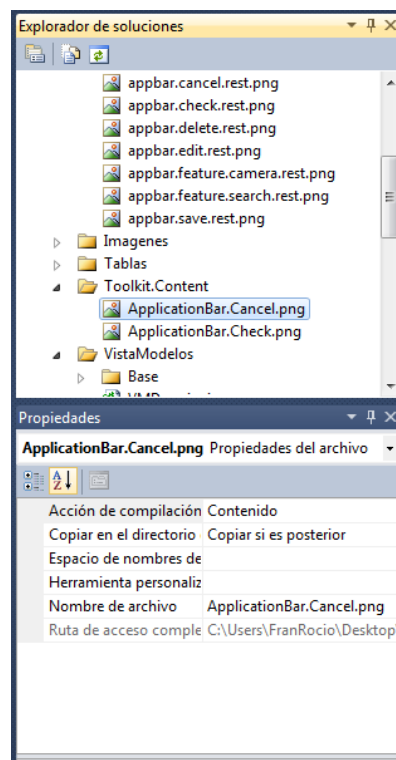


Ilustración 36. Propiedad Acción de compilación.

Si queremos inicializar el DatePicker con la fecha actual o cualquier otra fecha que no sea fija, tenemos que hacerlo en el constructor de la vista modelo, puesto que antes de mostrarlos tendremos que recuperar la fecha. Inicializar con la fecha actual es muy sencillo, tan sólo tenemos que llamar a la propiedad pública *Now*, que devuelve un *DateTime* con la fecha y hora actual. A continuación calculamos la fecha del día siguiente gracias al método *AddDays(int d)*, al cual hay que pasar cómo parámetros tantos días como queremos aumentar el *DateTime* con el que es llamado. Una vez calculada la fecha del día siguiente la asignamos a su variable correspondiente:

```
private DateTime fechaInicial;

private DateTime fechaRegreso;

public VMNuevoPlan()
{
    fechaInicial = DateTime.Now;
    today = DateTime.Now;
    tomorrow = today.AddDays(1);
    fechaRegreso = tomorrow;
}
```

A continuación explicamos cómo recuperar el valor que introduzca el usuario en el DatePicker. Para ello usaremos el enlace a datos con sus correspondientes *bindings*. El código XAML de uno de los *DatePickers* del formulario es:

```
<toolkit:DatePicker Name="fecIni" Width="170" Value="{Binding FechaInicial, Mode=TwoWay}"/>
```

Y el código correspondiente en la clase de la vista modelo es:

```
public DateTime FechaInicial
{
    get
    {
        return fechaInicial;
    }
    set
    {
        fechaInicial = value;
        NotifyChanged("FechaInicial");
    }
}
```

c) Vista modelo de la página NuevoPlan.xaml: interacción con la BD

Para que nada más mostrarse esta página el usuario vea los elementos que permiten seleccionar fechas correctamente rellenos hemos inicializado ambos elementos en el constructor de la vista modelo. Acabamos de ver cómo hacerlo con los elementos DatePicker que van a permitir seleccionar una fecha. El resto de campos de formularios los inicializaremos igualmente en el constructor de la vista modelo:

```
public VMNuevoPlan()
{
    nombre = "";
    presupuesto = 0;
}
```

El formulario consta de cuatro campos. Todos están conectados con la vista modelo mediante la técnica de enlace a datos y sus correspondientes *bindings*.

Código XAML de uno de los campos del formulario:

```
<TextBox Name="txtNombre"
    InputScope= "Text"
    Text="{Binding Nombre, Mode=TwoWay}"
</TextBox>
```

Código correspondiente en la clase de la vista modelo:

```
public DateTime FechaInicial
{
    get
    {
        return fechaInicial;
    }
    set
    {
        fechaInicial = value;
        NotifyChanged("FechaInicial");
        ComandoCrear.RaiseCanExecuteChanged();
    }
}
```

Hemos añadido en el *set* de todos los datos, la llamada al método *RaiseCanExecuteChanged()* de nuestro comando *ComandoCrear*, así refrescamos la interfaz y mostramos los cambios. Se considera cambio porque en el momento de crear el elementos del formulario se crean con los campos en blanco, y es en la vista modelo donde cambiamos el campo que viene por defecto por el valor inicial que nosotros le queramos dar, todo esto sucede antes de que se llegue a mostrar en pantalla por lo que pasa inadvertido para el usuario.

El comando *ComandoCrear* se lanza con el botón “Aceptar” de la barra de aplicaciones. Este botón tiene la novedad de que no se habilita hasta que no se cumplen una serie de requisitos en el formulario:

- El nombre no debe estar vacío ya que en la base de datos hemos puesto este campo como obligatorio.
- La fecha de salida debe ser igual o posterior a la actual.
- La fecha de regreso debe ser igual o posterior a la fecha de salida.
- El presupuesto ha de ser mayor que cero.

Aclarar que en esta aplicación se ha considerado la posibilidad de realizar viajes de un día por lo que la fecha de salida y la de regreso pueden ser la misma.

Estos requisitos se evalúan en el método *PuedeEjecutarCrearPlan()*. Como vimos anteriormente, este método devuelve un *booleano* con valor *true* en el caso de que se pueda ejecutar el comando y *false* en el caso que queramos que éste se deshabilite:

```
public bool PuedeEjecutarCrearPlan() {
    bool dev = true;
    string vacio= "";
    if (nombre == vacio || presupuesto <1) { dev = false; }
    if (fechaInicial.Date < DateTime.Now.Date)
    {
        MessageBox.Show("Seleccione una fecha de salida igual o posterior a la actual");
        dev = false;
    }
    else
    if (fechaInicial.Date > fechaRegreso.Date)
    {
        dev = false;
    }
    return dev;
}
```

En este método se evalúan los requisitos para habilitar el botón “Aceptar”, de tal forma que lo primero es comprobar si la variable *nombre* está vacía y si el presupuesto seleccionado es mayor que 0. En segundo lugar comprobamos que la fecha de salida del viaje no sea anterior a la fecha actual y en tercer lugar comprobamos que la fecha de regreso del viaje sea posterior o igual a la fecha de salida.

Una vez que se cumplen todos los requisitos y el método *PuedeEjecutarCrearPlan()* nos devuelve un valor *true*, accedemos al método *EjecutarCrearPlan()*, en el cual tenemos que insertar nuevos registros en la base de datos, tanto en la tabla “Viajes”, como en la tabla “Dias”.

Lo primero que hacemos en este método es calcular la duración del viaje y guardar su valor en una variable de tipo entero:

```
public void EjecutarCrearPlan()
{
    int diferenciaEnDias = calculaDuracion(fechaInicial.Date, fechaRegreso.Date);
```

A continuación abrimos la conexión a la base de datos. Creamos una lista de elementos de la clase “Viaje” e insertamos el nuevo “Viaje” creado mediante el método *Add()*. Simplemente tenemos que ir asignando cada valor recogido de la interfaz de usuario a su respectivo campo en la clase “Viaje”, para que luego se mapee a la base de datos:

```
using (AppDbContext context = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
{
    List<Viaje> nuevoPlanificacion = new List<Viaje>();
    nuevoPlanificacion.Add(new Viaje()
    {
        Nombre = nombre,
        FechaIni = fechaInicial.Date,
        FechaFin = fechaRegreso.Date,
        Presupuesto = presupuesto,
        Duracion = diferenciaEnDias,
        Dias = new System.Data.Linq.EntitySet<Dia>()
    });
```

Lo siguiente es añadir estos datos a la propiedad “Viajes” de nuestra clase *AppDbContext* usando el método *InsertAllOnSubmit*. Este método añade a la propiedad “Viajes” los registros que hemos creado con un estado de pendiente inserción. Por último nos queda indicar a la clase *AppDbContext* que envíe los cambios a la base de datos usando el método *SubmitChanges*:

```
context.Viajes.InsertAllOnSubmit(nuevoPlanificacion);
context.SubmitChanges();
```

Ahora tenemos que hacer lo propio para crear los registros en la tabla “Dias”, para ello nos creamos una lista con elementos de tipo “Dia” y recorremos un bucle *for* tantas veces cómo días dura el viaje, en cada iteración añadimos un nuevo “Dia” a la lista, todos ellos con el mismo identificador de viaje para indicar así, que pertenecen al mismo:

```
List<Dia> nuevosDias = new List<Dia>();
for (int i = 1; i <= diferenciaEnDias; i++)
{
    nuevosDias.Add(new Dia()
    {
        ViajeId = nuevoPlanificacion.ElementAt(0).ViajeId,
        NumDia = i,
        Descripcion = "... "
    });
}
context.Dias.InsertAllOnSubmit(nuevosDias);
context.SubmitChanges();
```

Por último llamamos al controlador de navegación para que nos devuelva a la página desde la que veníamos, es decir, Planificacion.xaml:

```
controlador.ControladorDeNavegacion.Current.BackTo();  
}
```

d) Método para calcular periodos de tiempo

Al principio del método que crea una planificación hemos tenido que calcular la duración del viaje. Para ello nos hemos creado un método en esta misma clase llamado *calculaDuracion(DateTime salida, DateTime regreso)* que nos devuelve un entero indicando el número de días:

```
public int calculaDuracion(DateTime salida, DateTime regreso)  
{  
    TimeSpan ts = regreso - salida;  
    return ts.Days+1;  
}
```

En este método, lo que hacemos es restar la fecha de regreso con la fecha de salida y lo guardamos en una variable de tipo *TimeSpan*, la cual almacena el tiempo en horas, días y minutos. Por último, nos quedamos sólo con los días y le sumamos uno ya que vamos a considerar en esta aplicación que el día de salida es también un día a planificar dentro del viaje.

e) Teclado numérico

A lo largo de toda la aplicación, cada vez que le pidamos al usuario que introduzca información se le mostrará un teclado para ello. Además, en los casos que tenga que introducir texto se le mostrará una ayuda para autocompletar palabras que saldrá en pantalla en la parte superior del teclado.

Para invocar los distintos tipos de teclados tenemos la propiedad *InputScope* de los *TextBox*. Usaremos el valor *Text* para introducir texto normal, *Maps* para introducir localizaciones y *Number* para introducir números (Ilustración 37):

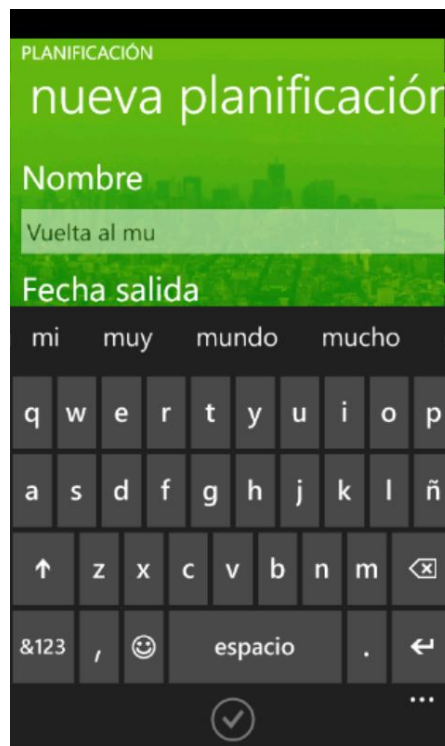


Ilustración 37. Teclado con ayuda en la escritura.

Para simplificar el cálculo numérico en la aplicación, hemos implantado una restricción a la hora de introducir cifras. Se ha eliminado la posibilidad de introducir números decimales. De esta forma vemos cómo podemos controlar lo que el usuario introduce y lo que no, en este caso le vamos a impedir introducir puntos decimales. Para ello hemos tenido que añadir algo más de código en el *code behind*. El resultado final es que el usuario al pulsar el punto del teclado numérico no obtiene respuesta alguna. Lo primero que tenemos que hacer para implementar esta restricción es invocar a un teclado numérico desde la página XAML mediante la propiedad *InputScope*= "Number":

```
<TextBlock FontSize="36"
            Foreground="{StaticResource PhoneAccentBrush}"
            Text="Presupuesto" TextAlignment="Center">
</TextBlock>
<TextBox Name="txtPresupuesto" Text="{Binding Presupuesto, Mode=TwoWay}" InputScope=
"Number" KeyUp="txtPresupuesto_KeyUp" Width="170">
```

Lo segundo es capturar el evento *KeyUp* que se produce cuando pulsamos alguna tecla y lanzamos el método *txtPresupuesto_KeyUp()* que a su vez llama a *validarPresupuesto()*:

```
private void txtPresupuesto_KeyUp(object sender, KeyEventArgs e)
{
    ValidarPresupuesto((TextBox)sender, false);
}
```

En el método *ValidarPresupuesto()* nos creamos una variable que contendrá los caracteres que queremos evitar que se introduzcan y a continuación le indicamos al *TextBox* que cada vez que detecte que se pulsa este carácter lo cambia por una cadena vacía, de esta manera el usuario detectará que el botón correspondiente al símbolo “.” está deshabilitado:

```
private void ValidarPresupuesto(TextBox textBoxControl, bool
permiteDecimales)
{
    String caracteresInvalidos = ".";
    textBoxControl.Text = textBoxControl.Text.Replace(caracteresInvalidos,
string.Empty);
    textBoxControl.SelectionStart = textBoxControl.Text.Length;
}
```

f) Tombstoning

Como ya explicamos en el tema del ciclo de vida de una aplicación, cuando el usuario la abandona, ya sea porque presiona el botón inicio o por lanzar un selector o lanzador del sistema, para sacar una foto, abrir un sitio web... recibimos el evento *Application_Deactivated*, donde deberemos guardar la información necesaria para que, una vez que el usuario vuelva, pueda continuar el uso de nuestra aplicación de manera normal. Así a lo largo de todas las páginas en las que el usuario cambie la información que se muestra por pantalla, tenemos que tener cuidado de guardarla siempre, antes de recibir el evento *Application_Deactivated*. Por ejemplo, en la pantalla que acabamos de ver, el usuario ha de introducir un nombre, un presupuesto y dos fechas. En cualquier momento entre que empieza a introducir información y le da al botón de guardar podemos recibir el evento *Application_Deactivated*, por ejemplo porque entra una llamada. En tal caso cuando se vuelva a la aplicación queremos que se le muestre al usuario la pantalla según la tenía, con los mismos datos que había introducido antes de recibir la llamada. De esta forma, tenemos dos escenarios donde conservar información: A nivel de página y a nivel de aplicación.

Antes de recibir el evento *Application_Deactivated*, recibiremos uno en nuestra propia página, el evento *OnNavigatedFrom*, que se ejecuta cada vez que abandonamos una página (en la página 60 explicamos este evento).

- **Tombstoning en páginas:** En primer lugar tenemos nuestra página activa, que debe conservar la información relativa a la misma que le permita restaurarse y continuar operando. En esta página

usaremos los métodos *OnNavigatedTo* y *OnNavigatedFrom* para persistir información y recuperarla.

Se usa el diccionario *State* del objeto *PhoneApplicationPage* para persistir nuestra información. Internamente se usa un serializador *DataContractSerializer* para guardar los datos, por lo que añadimos una referencia al ensamblado *System.Runtime.Serialization.dll* en nuestro proyecto y un *using* al *namespace System.Runtime.Serialization*. Añadimos en nuestra vista modelo el atributo *[DataContract]* y sus propiedades el atributo *[DataMember]*:

```
[DataContract]
public class VMNuevoPlan : VMBase
{
    .
    .
    .

}

[DataMember]
public string Nombre
{
    .
    .
    .

[DataMember]
public DateTime FechaInicial
{
    .
    .
    .
}
```

A continuación necesitamos introducir código en nuestra vista que gestione la persistencia de la información.

Lo primero que hemos hecho es añadir una variable en la clase que controle cuándo estamos trabajando con una nueva instancia y cuándo estamos recuperándonos del *tombstoning*. Y una variable también en la clase, que contendrá nuestra vista modelo:

```
public partial class NuevoPlan : PhoneApplicationPage
{
    VMNuevoPlan contexto;
    bool newInstance = true;
```

En el constructor de nuestra clase simplemente vamos a establecer la variable *newInstance* a *true*. Esto se debe a que cuando nos recuperamos del *tombstoning* se vuelve a ejecutar el constructor de la página, mientras que si nuestra aplicación está “durmiente” no se ejecutará:

```
public NuevoPlan()
{
    InitializeComponent();
    newInstance = true;
}
```

Ahora vemos cómo, al sobrescribir el método *OnNavigatedFrom* de nuestra página, podemos guardar la información:

```
protected override void
OnNavigatedFrom(System.Windows.Navigation.NavigationEventArgs e)
{
    if (e.NavigationMode != NavigationMode.Back)
    {
        this.State["VMNuevoPlan"] = contexto;
    }
}
```



```
}
```

Lo primero que hemos hecho es comprobar si el usuario ha pulsado el botón “atrás”. Si este es el caso no necesitaremos conservar información, puesto que la página se va a descargar de memoria y será destruida junto con la colección *State* que le pertenece.

Si no estamos navegando hacia atrás, significa que estamos saliendo de la página. Ya sea a otra página o porque el usuario haya presionado el botón inicio, este es el momento de guardar nuestra vista modelo dentro del diccionario *State* de nuestra página, el cual será serializado y preservado de forma automática.

Por último se debe ser capaz de recuperar los datos guardados o, si nos encontramos en una nueva instancia (la primera vez que se carga la página) poder crear una nueva instancia de nuestra vista modelo. Esto lo haremos sobrescribiendo el método *OnNavigatedTo*:

```
protected override void
OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
    if (newInstance)
    {
        if (contexto == null)
        {
            if (State.Count > 0)
            {
                contexto = (VMNuevoPlan)State["VMNuevoPlan"];
            }
            else
            {
                contexto = new VMNuevoPlan();
            }
        }
        this.DataContext = contexto;
    }
    newInstance = false;
}
```

Lo primero que necesitamos es saber si nos encontramos en una nueva instancia y si nuestra variable vista modelo es nula. Si se cumplen estas dos situaciones significa que estamos ante una página que se está instanciando en estos momentos.

A continuación comprobamos si nuestra variable *State* contiene alguna clave. Si es así, significa que tenemos información guardada y simplemente la recuperamos. De lo contrario significa que no hemos guardado información, por lo que solo nos queda crear una nueva instancia de nuestra vista modelo y, finalmente, asignarla al *DataContext* de nuestra página.

Si recordamos el esquema del ciclo de vida, en Windows Phone 7.5, nuestra aplicación queda en un estado “durmiente” al salir del menú inicial. Solo se realiza el *tombstoning* si el dispositivo se queda sin memoria. Ejecutando en el emulador, es muy complicado que esto ocurra, por lo que si probamos nuestra aplicación en Visual Studio 2010, presionamos el botón de inicio y luego volvemos a la aplicación, veremos que no vuelve a lanzar el constructor de la página en la que nos encontremos. Por lo tanto, el evento *OnNavigatedTo* no ejecuta ningún código, aunque sí que se muestran los datos que hayamos introducido de manera correcta, debido al estado durmiente.

Podemos forzar a que esto no ocurra con una opción dentro de las propiedades del proyecto, en la pestaña “Depurar” (Ilustración 38):

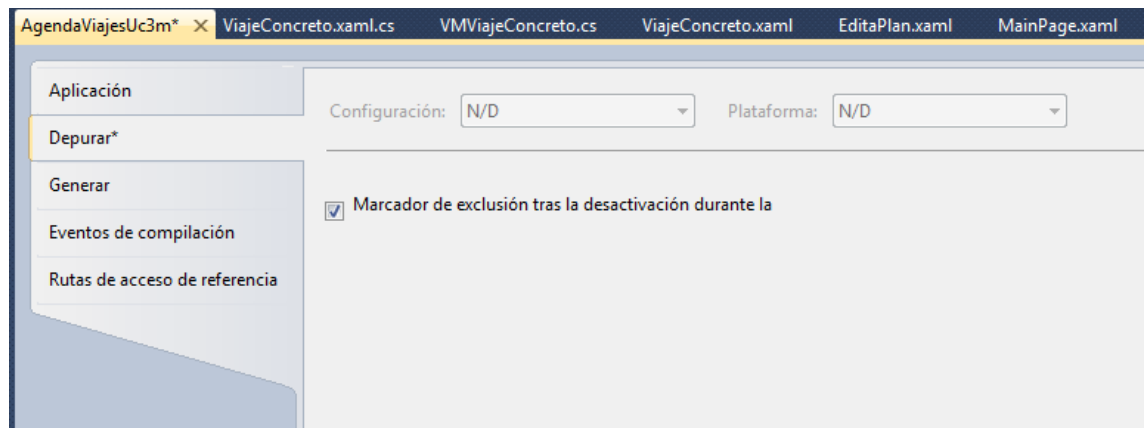


Ilustración 38. Activación del *tombstoning* automático.

Marcando la opción “Marcador de exclusión tras la desactivación durante la ejecución” obligamos a nuestra aplicación a realizar la desactivación y el *tombstoning*. Si volvemos a probar, veremos que ahora sí, cuando volvemos a nuestra aplicación se invoca al constructor y después a nuestro método *OnNavigatedTo* que se encarga de establecer los datos correctos. Es muy importante recordar que, aunque todas las aplicaciones pueden pasar a estado durmiente, no es seguro que esquivemos el *tombstoning*. Es por eso que, aun sabiendo que inicialmente no se realizará, siempre guardemos los datos, pues el *tombstoning* no nos avisará. Simplemente nuestra aplicación será destruida y sacada de memoria y todos los datos no persistidos se perderán sin ninguna opción de recuperación.

- **Tombstoning en aplicación:** En segundo lugar tenemos el evento *Application_Deactivated*. Este permitirá guardar todos los datos comunes a nuestra aplicación, como datos descargados desde un servicio web que se usen en varias páginas.

Por datos comunes entendemos que son todos aquellos que sean necesarios para el funcionamiento de múltiples páginas. Si delegásemos su persistencia a las páginas acabaríamos con varias copias de los datos dentro del diccionario *State* de cada página. Esto supondría un gasto de memoria y almacenamiento y podría provocar que cada página tuviese distintas versiones de los mismos datos.

Para solucionar esto, en nuestras páginas ignoramos la persistencia de estos datos comunes y dejamos que sea el evento *Application_Deactivated* el que se encargue de gestionar la persistencia de los datos con un ámbito de aplicación. En esta aplicación no vamos a tener este tipo de datos por lo que en el evento *Application_Deactivated* solo vamos a tener una sentencia que explicamos a continuación:

```
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    PhoneApplicationService.Current.ApplicationIdleDetectionMode =
        IdleDetectionMode.Enabled;
}
```

Existe una propiedad dentro de la instancia *Current* de *PhoneApplicationService* llamada *ApplicationIdleDetectionMode* que nos permite hacer que nuestra aplicación siga ejecutándose aunque apaguemos la pantalla del teléfono.

Esta propiedad admite dos valores: *Disabled*, caso en el que no detectará que la pantalla ha sido bloqueada y la aplicación seguirá ejecutándose o *Enabled*, que es el valor escogido en la mayoría de las ocasiones y cerrará la aplicación al bloquear / apagar la pantalla del dispositivo.

Tenemos que tener en cuenta que esta característica permite que nuestra aplicación siga funcionando, por lo que seguiremos consumiendo batería, un bien muy preciado en los actuales dispositivos móviles.

Con Windows Phone 7.5, Microsoft pide que se sea muy consciente del uso de esta característica. Solo debemos usarla en casos excepcionalmente necesarios para el funcionamiento de nuestra aplicación, como por ejemplo, una aplicación de GPS que monitorice carreras o entrenamiento y con la que necesitemos un goteo constante de información de posición.

X. Pantalla de edición de una planificación: *Pivot*, mapas

Terminábamos el anterior apartado pulsando el botón “Aceptar” para crear una planificación. Ahora nos situamos en la página *Planificiacion.xaml*. En la cual tenemos una lista compuesta por las distintas planificaciones de viajes que ha creado el usuario. Este tiene la posibilidad de seleccionar una planificación y, o bien eliminarla, o editarla. Vamos a empezar viendo la última opción, en la que nuestra aplicación navegará hasta la página de edición.

Como novedades de esta página, destaca la utilización de un nuevo elemento de diseño de páginas, el *Pivot*. Esto implicará que tengamos que mostrar u ocultar la barra de aplicación según las vistas del *Pivot*, además utilizaremos los potentes mapas de Bing en la tercera vista:

- a. *EditaPlan.xaml*: Primera vista del *Pivot*.
- b. *EditaPlan.xaml*: Segunda vista del *Pivot*.
- c. *EditaPlan.xaml*: Tercera vista del *Pivot*.

a) *EditaPlan.xaml*: Primera vista del *Pivot*

El elemento *Pivot* es la novedad en el diseño de esta página. Este elemento permite mostrar al usuario distintas vistas a las que puede acceder con un simple gesto de arrastrar el dedo sobre la pantalla hacia la derecha o izquierda, o presionando sobre los diferentes encabezados que muestre. El *Pivot* basa su existencia en mostrar datos de forma eficiente al usuario. Tenemos un pequeño título en la parte superior, que siempre permanece estático y una lista horizontal con las diferentes vistas de las que disponemos, formadas por elementos *PivotItem*, el resto de la pantalla lo ocupará el contenido de cada *PivotItem*.

Cada uno de estos elementos *PivotItem* actuará como una vista de nuestra página, conteniendo todos los elementos que necesitemos para crear la interfaz de usuario (Ilustración 39).

El elemento *Pivot* se compone de dos capas:

- **Lista de cabeceras:** Se trata de la parte superior del elemento *Pivot*. Además de mostrar el título que escojamos para la página, también se encargará de mostrar el texto que establezcamos como cabecera de cada *PivotItem*.
- ***PivotItem Presenter*:** Un panel *ItemPresenter* estándar de Silverlight, encargado de mostrar cada elemento *PivotItem* cuando es seleccionado.

El elemento *PivotItem* muestra el contenido que definamos para cada vista de nuestra página. El contenido de cada *PivotItem* se define en la propiedad *Content*, un elemento *ContentPresenter* estándar de Silverlight que admite contenido estructurado en un elemento contenedor como una *Grid*, un *StackPanel*, etc.

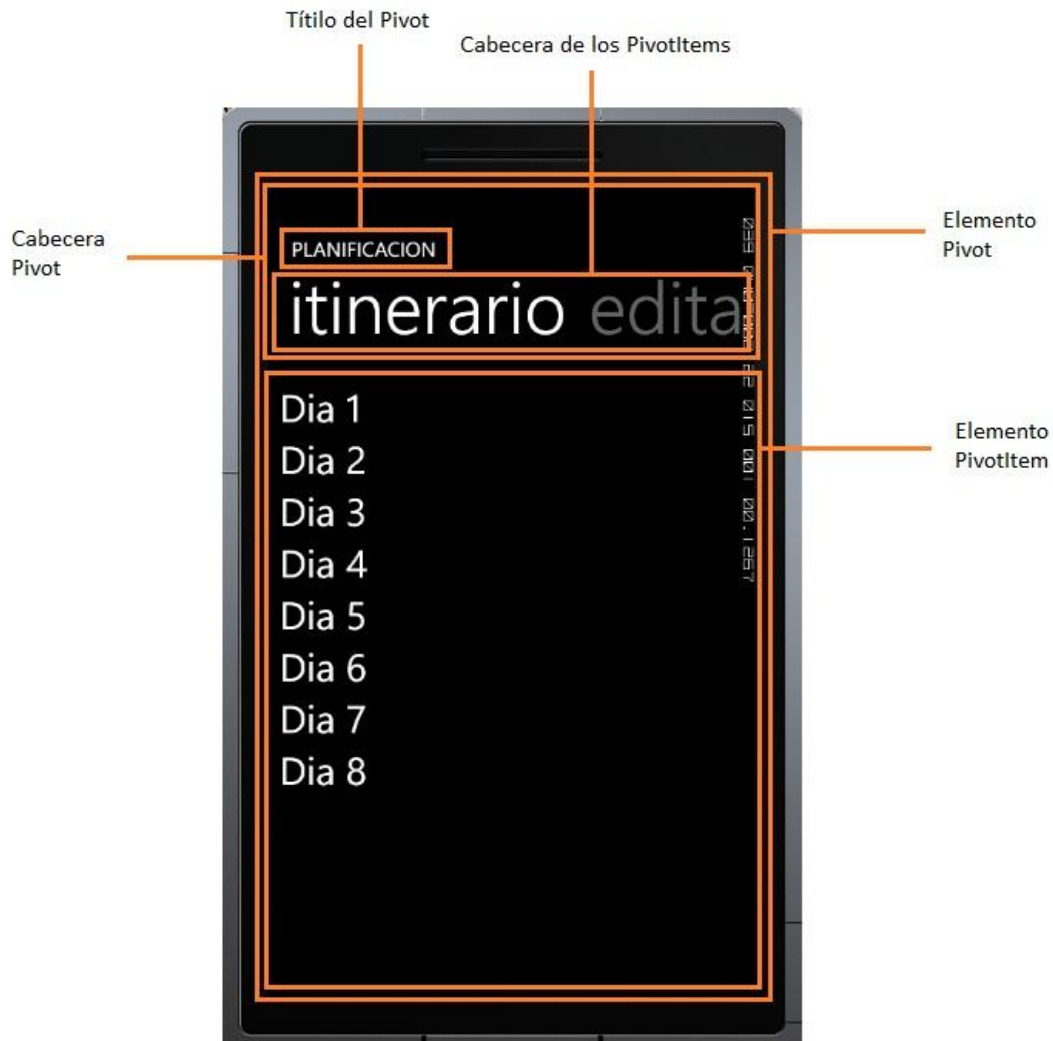


Ilustración 39. Detalle del elemento Pivot.

Para trabajar con el elemento *Pivot*, simplemente necesitamos añadir una referencia en nuestro proyecto al ensamblado *Microsoft.Phone.Controls*. A continuación necesitamos incluir ese ensamblado y su *namespace* en nuestra página XAML:

```
xmlns:controls="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"
```

Al definir el *namespace* y el ensamblado que queremos usar, ya podemos incluir un elemento *Pivot* en nuestra página y a continuación, directamente dentro del elemento *Pivot*, definir las diferentes vistas, usando el elemento *PivotItem*:

```
<controls:Pivot Title="PLANIFICACION" Name="pivotEditor"
    LoadingPivotItem="cargandoPivotItem">
    <controls:Pivot.TitleTemplate>
        <DataTemplate>
            <TextBlock Text="PLANIFICACION" Foreground="White" FontSize="20"
                FontFamily="Segoe WP"></TextBlock>
        </DataTemplate>
    </controls:Pivot.TitleTemplate>
    <controls:PivotItem Name="lista">
        <controls:PivotItem.Header>
            <TextBlock Text="itinerario"
```

```

        Foreground="White"
        FontSize="72">
    </TextBlock>
</controls:PivotItem.Header>
<ListBox Name="listaDias" Style="{StaticResource estiloListBox}"
        SelectedItem="{Binding ItemSeleccionado, Mode=TwoWay}"
        SelectionChanged="lstDias_SelectionChanged"
        ItemsSource="{Binding ListaDias, Mode=TwoWay}" >

    </ListBox>
</controls:PivotItem>
</controls:Pivot>

```

En este caso, hemos añadido un elemento *ListBox* en la primera vista de nuestro Pivot, este *ListBox* va a contener una lista con tantos elementos como días dure el viaje seleccionado. Así mediante el enlace a datos podemos acceder a las bases de datos para saber los días de los que consta el viaje (Ilustración 40):



Ilustración 40. Primera vista del Pivot: Itinerario.

Para saber qué planificación queremos editar, y por tanto recuperar correctamente sus datos, en la anterior página pasábamos un parámetro en el cual almacenábamos el identificador de viaje, de manera que lo primero que tenemos que hacer al navegar a esta página es obtenerlo.

Para ello en el constructor del *code behind* de la página *EditaPlan.xaml*, simplemente inicializamos los componentes y ponemos a *true* la variable que indica que se trata de una nueva instancia de la página. Es en el método *OnNavigatedTo()* donde rescatamos el parámetro.

Dentro del método *OnNavigatedTo()*, además de implementar la funcionalidad para poder rescatar los datos, si se produce *tombstoning*, lo que hacemos es rescatar el parámetro llamado "ID". Éste viene como cadena de texto y por tanto tenemos que transformarlo en entero para pasárselo al constructor de la vista modelo *VMEditaPlan.cs*:

```

string aux = NavigationContext.QueryString["ID"];
id = Convert.ToInt16(aux);
contexto = new VMEditaPlan(id);

```

Por último, después de crear nuestra vista modelo, tenemos que establecerla como la fuente de datos de nuestra vista:

```
this.DataContext = contexto;
```

b) EditaPlan.xaml: Segunda vista del *Pivot*

En la segunda vista de nuestro *Pivot* vamos a tener el mismo formulario que en la página NuevoPlan.xaml ya que así damos la oportunidad de cambiar los datos de la planificación al usuario. Al final de este formulario tenemos una barra de aplicación con los botones de “Guardar” y de “Cancelar” con la novedad de que sólo se muestra este elemento cuando nos situamos en esta vista del *Pivot*, de manera que si nos desplazamos a la izquierda o la derecha, la barra de aplicación volverá a ocultarse (Ilustración 41):

A screenshot of a mobile application interface. The background is a green-tinted image of a city skyline. At the top, the word 'PLANIFICACION' is written in small white capital letters. Below it, the title 'editar una plan' is displayed in large white font. The form contains four input fields: 'Nombre' with the value 'Lisboa', 'Fecha inicial' with '17/05/2013', 'Fecha final' with '19/05/2013', and 'Presupuesto' with '380'. At the bottom, there is a dark grey application bar with two circular icons: a square with a plus sign and a circle with an 'x'. To the right of these icons are three small white dots.

Ilustración 41. Segunda vista del *Pivot*. Formulario.

Para conseguir tal efecto hemos controlado el evento *LoadingPivotItem* del elemento *Pivot* con una condición. Solo en el caso de que el *PivotItem* activo sea la segunda vista, cuyo nombre es “editar”, mostraremos la barra de aplicaciones:

```
private void cargandoPivotItem(object sender, PivotItemEventArgs e)
{
    if (e.Item == editar)
        ApplicationBar.IsVisible = true;
    else
        ApplicationBar.IsVisible = false;
}
```

c) EditaPlan.xaml: Tercera vista del *Pivot*, mapas

Por último, en la tercera vista del *Pivot*, hemos incluido un mapa en el que situamos el centro en función del nombre que tenga la planificación. Complementando al mapa, hemos incluido en la pantalla un *TextBox* en el que podemos realizar búsquedas, los correspondientes botones para acercar y alejar el zoom y un control tipo interruptor que consta de dos posiciones: *On* y *Off*. Usaremos este interruptor para

cambiar el tipo de vista del mapa. Para hacer esto posible usaremos la propiedad *Mode* del tipo *MapMode*. De esta clase base *MapMode* derivan las clases *RoadMode*, *AerialMode* y *MercatorMode*:

- *RoadMode* muestra una vista de carreteras, sin información ni imagen geográfica.
- *AerialMode* muestra una vista aérea del terreno. Podemos escoger entre mostrar o no las etiquetas de las calles y lugares de interés.
- *MercatorMode* nos muestra un mapa vacío, sin la capa base del terreno sobre el cual podemos crear nuestros propios mapas.

A la hora de configurar el interruptor tendremos que tener en cuenta cómo queremos que se muestre por defecto, en estado *ON* o en estado *OFF*. Para ello tenemos la propiedad *IsChecked* que la establecemos a “false” para dejarla en estado *OFF*. Finalmente tenemos que relacionar cada estado con un tipo de vista. Para el estado *OFF* modificaremos la propiedad *Unchecked* poniendo el evento que queremos lanzar, *aereo_click*. Mientras que para el estado *ON* modificaremos la propiedad *Checked*, que lanzará el evento *carreteras_click*.

Para poder acceder a las últimas clases derivadas de *MapMode*, añadimos en el archivo *EditaPlan.xaml.cs* el *using* al *namespace Microsoft.Phone.Controls.Maps*. A continuación vemos el código subyacente de los anteriores eventos que realizan la tarea de cambiar el tipo de mapa.

```
private void aereo_click(object sender, RoutedEventArgs e)
{
    mapa1.Mode = new AerialMode(true);
}

private void carreteras_click(object sender, RoutedEventArgs e)
{
    mapa1.Mode = new RoadMode();
}
```

La inserción del elemento *Bing Maps* es sencillo, lo primero que necesitamos es ir a la caja de herramientas de Visual Studio y en los controles de Windows Phone encontramos uno llamado *Map* (Ilustración 42):

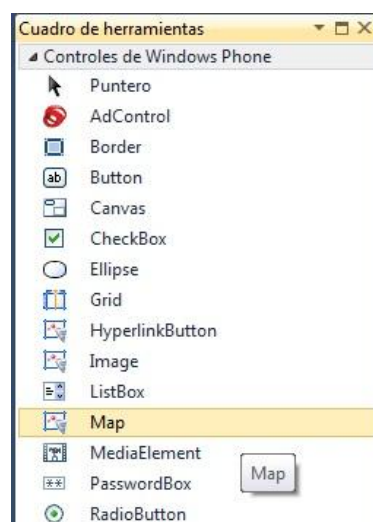


Ilustración 42. Elemento Bing Maps en la caja de herramientas.

Simplemente debemos arrastrarlo a nuestra página y el mismo elemento se encargará de añadir una referencia en nuestro proyecto al ensamblado *Microsoft.Phone.Controls.Maps* y de añadirlo a los *namespaces* de nuestra página (Ilustración 43):

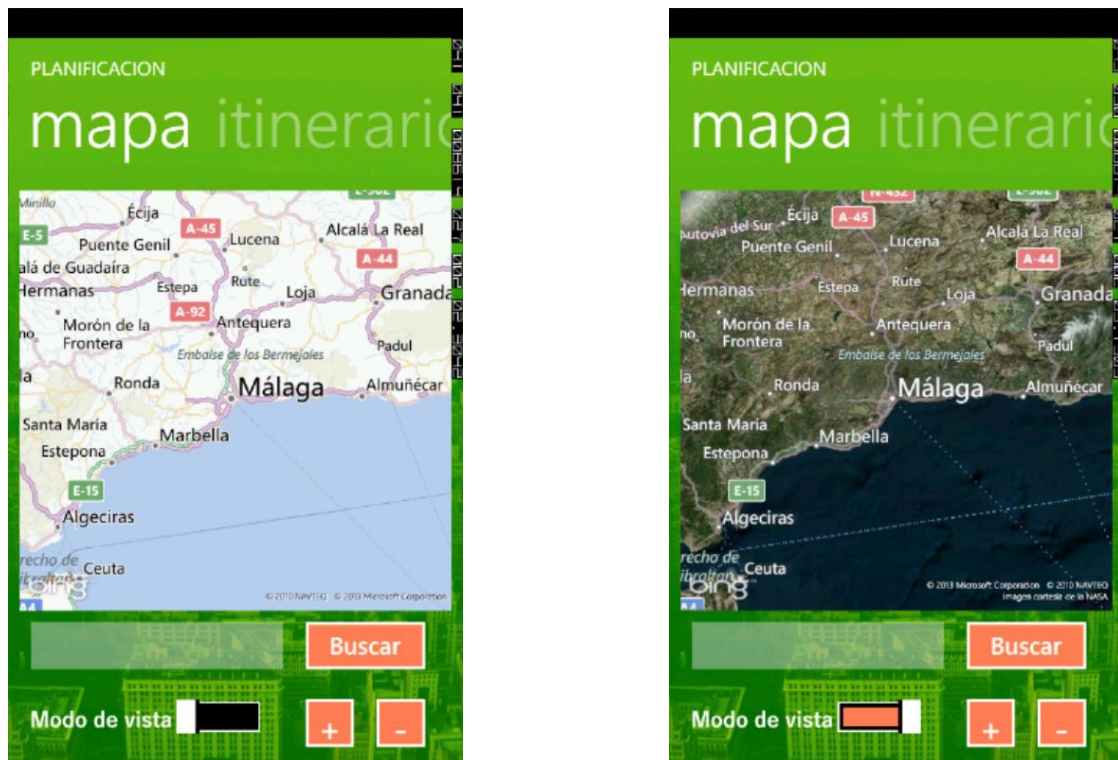


Ilustración 43. Tercera vista del Pivot: Mapa.

d) Vista modelo de la página *EditaPlan.xaml*: servicios web de Microsoft, geolocalización, recuperar datos de la BD

Vamos a ver a continuación cómo sería la vista modelo de esta página. Al componerse ésta de tres vistas contendrá bastantes métodos, entre los que destacan el que nos redirige a la página *Descripcion.xaml* según el día que hayamos seleccionado, el que guarda los cambios hechos por el usuario o los métodos que permiten dotar al mapa de mayor usabilidad. Entre estos últimos veremos cómo consumir alguno de los servicios web que proporciona Microsoft.

- i. El constructor.
- ii. Método `calcularDias()`.
- iii. Selección de día para añadir una descripción.
- iv. Botón guardar cambios.
- v. El buscador del mapa.
- vi. Aumentar y disminuir el zoom.

i. El constructor

El constructor de esta página tiene un papel fundamental, puesto que en él inicializamos todos los componentes que se le muestran al usuario mediante el enlace a datos. Lo primero que vamos a hacer es tratar el parámetro que hemos rescatado para hacer una búsqueda en la base de datos. Así, únicamente tenemos que hacer un *Select* para tener en la lista “listaViajes” el “Viaje” al que le corresponde el “ID” del parámetro:

```
listaViajes = (from Viaje viaje in contextDb.Viajes where viaje.ViajeId == parametro
select viaje).ToList();
```

El siguiente paso es rellenar los campos del formulario de la segunda vista del *Pivot* para mostrárselos al usuario por si quiere modificar alguno o simplemente consultar:


```

Nombre = listaViajes.ElementAt(0).Nombre;
Presupuesto = listaViajes.ElementAt(0).Presupuesto;
FechaInicial = listaViajes.ElementAt(0).FechaIni;
FechaRegreso = listaViajes.ElementAt(0).FechaFin;

```

A continuación llamamos al método que hemos creado en esta misma clase llamado *calcularDias()*, que crea una lista con tantos elementos como días consta el viaje. Por último situamos el centro del mapa y el nivel de zoom. Para ello llamamos al método *calcularCoordenadas(string nombre)* pasándole como parámetro el nombre del viaje, mientras que el nivel de zoom lo establecemos en 8. Se trata de un nivel intermedio puesto que el nivel de zoom puede tomar valores entre 1 y 16:

```

calcularDias();
calcularCoordenadas(listaViajes.ElementAt(0).Nombre);
ZoomLevel = 8;

```

ii. Método *calcularDias()*

En esta ocasión hemos creado el método porque tenemos que utilizar el mismo código en varias partes dentro de la vista modelo. Este método calcula los días de los que consta el viaje y los incluye en la lista que esta enlazada con el *ListBox* de la primera vista de la presentación.

Con un *Select* seleccionamos los días correspondientes a este viaje para posteriormente añadirles el *String* “Día” de forma que su visualización en el *ListBox* “ListaDias” en la primera vista del *Pivot* sea más clara:

```

public void calcularDias()
{
    Lista = (from Dia dia in contextDb.Dias where dia.ViajeId == parametro select
    dia).ToList();
    List<String> listaAux = new List<String>();
    for (int i = 0; i < Lista.Count; i++)
    {
        listaAux.Add("Dia " + Lista.ElementAt(i).NumDia);
    }
    ListaDias = listaAux;
}

```

iii. Selección de día para añadir una descripción

Aprovechando que en la primera vista del *Pivot* tenemos una lista con los días que dura el viaje, vamos a permitir al usuario que añada una descripción o comentario sobre lo que pretende hacer ese día. Cuando seleccione un día de la lista, la aplicación navegará a la página donde podrá añadir texto. El método que se encarga de capturar el evento que se produce cuando el usuario selecciona un día es *EjecutarSelectionChanged()*.

Antes de entrar en detalle en el código del método hay que explicar una pequeña modificación que se hizo en el constructor de la página. El problema surgía cuando el usuario volvía a la página *EditaPlan.xaml* después de visitar la página *Descripcion.xaml*. El día seleccionado para ser editado se quedaba en estado “seleccionado” por lo que si el usuario quería volver a la descripción de ese día no podía. Para solucionarlo añadimos la siguiente línea que se ejecuta en caso de que volvamos de la página *Descripcion.xaml*:

```

listaDias.SelectedIndex = -1;

```

Lo que hacemos con esa línea es resetear la selección, es decir, le pasamos el valor de índice de un elemento que no está en la lista, de esta manera libera al que había seleccionado para no seleccionar ninguno y dejarlo a la espera de que el usuario escoja otro día.

Al obligarle a cambiar de valor al atributo *ItemSeleccionado*, aunque sea a un valor equivocado, este hará que se ejecute el método *EjecutarSelectionChanged()*. Por lo tanto lo primero que hacemos en el método es comprobar que *ItemSeleccionado* tiene un valor nulo o no. En caso de que sea nulo, no hacemos nada

en el método y dejamos que la aplicación vuelva a la interfaz. En el otro caso, obtendríamos el día que el usuario ha seleccionado:

```
public void EjecutarSelectionChanged()
{
    if (ItemSeleccionado != null)
    {
        int idDiaSel;
        string selDia = ItemSeleccionado.ToString();
```

Como *ItemSeleccionado* es una cadena que mezcla letras y números, tenemos que trabajar con esa cadena para obtener solamente el número de día:

```
        selDia = selDia.Substring(4);
        int numDiaSel = Convert.ToInt16(selDia);
```

A continuación seleccionamos el identificador de día, conociendo de antemano el identificador de viaje (es decir, el parámetro que recibí desde la página anterior) y el número de día:

```
        using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
        {
            List<Dia> diaSel = (from Dia dia in contextDb.Dias where (dia.ViajeId
== parametro && dia.NumDia == numDiaSel) select dia).ToList();
            idDiaSel = diaSel.ElementAt(0).DiaId;
        }
```

Por último le pasamos el identificador de día a la siguiente pantalla en la que podremos editar ese día en concreto:

```
        string parameters = string.Format("IDDIA={0}", idDiaSel);
        controlador.ControladorDeNavegacion.Current.NavigateTo("Descripcion",
parameters);
```

iv. Botón “Guardar cambios”

La funcionalidad principal de esta página es poder editar la información relativa a la planificación de viaje. Esto lo puede hacer el usuario cambiando los valores del formulario de la segunda vista del *Pivot*. Una vez hechos los cambios podrá guardarlos pulsando el botón “Guardar cambios”. Este botón está conectado con la vista modelo mediante un comando:

```
public DelegateCommand ComandoGuardar
{
    get
    {
        if (comandoGuardar == null)
            comandoGuardar = new DelegateCommand(EjecutarGuardarPlan,
                                                    PuedeEjecutarGuardarPlan);
        return comandoGuardar;
    }
}
```

El cual ejecuta el método *EjecutarGuardarPlan()*, siempre que el método *PuedeEjecutarGuardarPlan()* devuelva un booleano con valor *true*.

Las condiciones para que devuelva *true* son las siguientes:

- Que el campo nombre no esté vacío y el presupuesto sea mayor que cero.
- Que la fecha inicial del viaje no sea menor que la fecha actual.
- Que la fecha inicial del viaje no sea mayor a la fecha de regreso.

Dentro del método *EjecutarGuardarPlan()* lo primero que hacemos es deshacernos del foco del objeto seleccionado, ya que no es hasta ese momento cuando se hace notificación del cambio y entra en funcionamiento el enlace a datos:

```
public void EjecutarGuardarPlan ()
{
    object focusObj = FocusManager.GetFocusedElement();
    if (focusObj != null && focusObj is TextBox)
    {
        var binding = (focusObj as
            TextBox).GetBindingExpression(TextBox.TextProperty);
        binding.UpdateSource();
    }
}
```

Si no hiciéramos esto, los distintos elementos del formulario no serían leídos a tiempo para ser guardados en la base de datos. De esta manera obligamos a que así sea. Este problema hay que tenerlo siempre presente a la hora de trabajar con la barra de aplicaciones. Otra posible solución hubiera sido colocar el botón “Aceptar” en un simple elemento *Button*. Él directamente al ser pulsado, obtiene el foco de esa página y no necesitaría código adicional para guardar los datos, pero romperíamos la estética Metro de nuestra aplicación.

A continuación calculamos la duración del viaje y abrimos la conexión con la base de datos:

```
int diferenciaEnDias = calculaDuracion(fechaInicial.Date, fechaRegreso.Date);
using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
{
```

Después, recuperamos el viaje con el que estamos trabajando para ir actualizando sus datos uno por uno. Es muy importante no olvidar guardar de nuevo el campo “ViajeId”, aun simplemente sobrescribiendo su valor. Al tratarse de la clave primaria, siempre tendremos que insertarla cuando hagamos cambios en una fila. De lo contrario nos devolverá el error “*error Row not found or changed*”:

```
int diferenciaEnDiasOld;
listaViajes = (from Viaje viaje in contextDb.Viajes where viaje.ViajeId == parametro
select viaje).ToList();
diferenciaEnDiasOld = listaViajes.ElementAt(0).Duracion;
listaViajes[0].ViajeId = parametro;
listaViajes[0].Nombre = Nombre;
listaViajes[0].FechaIni = FechaInicial.Date;
listaViajes[0].FechaFin = FechaRegreso.Date;
listaViajes[0].Duracion = diferenciaEnDias;
listaViajes[0].Presupuesto = Presupuesto;
contextDb.SubmitChanges();
```

También tenemos que actualizar la tabla de los días, pero antes de nada tenemos que comprobar si los días han aumentado, han disminuido o se mantienen el mismo número:

```
if (diferenciaEnDiasOld < diferenciaEnDias)
{
    int cuantosDiasNuevos = diferenciaEnDias - diferenciaEnDiasOld;
```

En caso de tener que ampliar el número de días, tenemos que ver cuántos días son, para construir un bucle en el que cada iteración añada un día en la base de datos:

```
for (int i = 1; i <= cuantosDiasNuevos; i++)
{
    List<Dia> nuevosDias = new List<Dia>()
    {
        new Dia() {ViajeId = parametro, NumDia = diferenciaEnDiasOld + i,
            Descripcion = "..."},
    };
    contextDb.Dias.InsertAllOnSubmit(nuevosDias);
    contextDb.SubmitChanges();
}
```

```
    }
}
```

En el caso de tener que borrar días de la base de datos, nos creamos un bucle del mismo tamaño que los días a borrar y dentro de él lo que hacemos es buscar el día a borrar en concreto en la base de datos, ponerlos en un estado pendiente de borrado y finalmente confirmar los cambios en la base de datos:

```
else if (diferenciaEnDiasOld > diferenciaEnDias)
{
    for (int i = diferenciaEnDias; i < diferenciaEnDiasOld; i++)
    {
        List<Dia> listaDiasAux = (from Dia dia in contextDb.Dias where
            (dia.ViajeId == parametro && dia.NumDia == (i + 1)) select
            dia).ToList();
        contextDb.Dias.DeleteAllOnSubmit(listaDiasAux);
        contextDb.SubmitChanges();
    }
}
```

Antes de salir del método mostramos un mensaje al usuario indicándole que los cambios se han guardado correctamente y llamamos al método *calcularDias()*, para actualizar la lista de la primera vista del *Pivot*:

```
MessageBox.Show("Los cambios han sido guardados correctamente");
calcularDias();
```

v. *Mapa, servicios de Microsoft y geolocalización*

En la tercera vista del *Pivot* teníamos un mapa con varias opciones en la parte inferior, la más interesante de ellas es la que permite localizar lugares directamente con un botón.

Para añadir dicha funcionalidad hemos enlazado el botón “Buscar” con un comando que llama al método *EjecutarBuscarMapa()* que a su vez llama al método *calcularCoordenadas(string nombre)*, método que ya vimos que se utilizaba en el constructor. En esta ocasión es llamado cada vez que el usuario introduce un lugar en el *TextBox* y hace clic en el botón “Buscar”.

En este método hacemos uso de los *Microsoft BingMaps SOAP Services* [6], esto es un conjunto de servicios web programables que permiten marcar direcciones en el mapa, buscar puntos de interés, integrar mapas e imágenes, devolver rutas entre dos puntos e incorporar otro tipo de localizaciones en la aplicación.

Los servicios web han sido contruidos usando *Windows Communication Foundation (WCF)*. Hay cuatro *Bing Maps SOAP Services*, de los cuales en esta aplicación solo hemos usado el primero: *Geocode Service*, *Imagery Service*, *Route Service*, y *Search Service*. A continuación describimos cada uno de los servicios junto son sus respectivos métodos [7]:

Geocode Service:

El servicio de geolocalización se puede utilizar para emparejar direcciones, lugares y elementos geográficos con su latitud y longitud en el mapa, así como devolver información para unas coordenadas específicas dando su latitud y longitud.

Los métodos del servicio de geolocalización son:

- *Geocode*: Devuelve una posición (latitud y longitud) dada una dirección o lugar.
- *Reverse Geocode*: Encuentra una dirección o lugar para una ubicación dada.

Imagery Service:

El Servicio de imágenes se puede usar para obtener imágenes de mapas, sus URIs u otros metadatos. Por ejemplo podemos enlazar a un mapa con un icono de señalización en una determinada ubicación o mostrar imágenes de mapa en modo Carretera o Satélite.

Bing Maps dibuja mapas usando una proyección ortográfica centrada en el punto central de cualquier mapa dado y basándose en el sistema de coordenadas WGS84.

Los métodos del servicio de imágenes son:

- *GetImageryMetadata*: Permite obtener los metadatos de una imagen solicitada, dando las coordenadas para su búsqueda.
- *GetMapUri*: Nos devuelve la URI a una imagen de mapa dadas unas coordenadas y un tamaño de imagen concreto.

Route Service:

El servicio de rutas genera rutas e indicaciones para la conducción a partir de localizaciones o puntos de interés. Por ejemplo, se pueden obtener indicaciones incluyendo advertencias y sugerencias de rutas de tráfico entre múltiples ubicaciones.

Los métodos del servicio de rutas son:

- *CalculateRoute*: Calcula una ruta entre dos puntos aportando las indicaciones y otros datos de ruta.
- *CalculateRoutesFromMajorRoads*: Calcula los puntos de partida o indicaciones hacia una posición desde las principales carreteras cercanas. (No disponible en España).

Search Service:

Este servicio es útil para buscar lugares de interés en una dirección o ubicación: “sushi; Plaza América”. En España este servicio no devuelve resultados.

Este servicio solo tiene un método disponible:

- *Search*: Devuelve los resultados a una consulta hecha con *Strings*.

Para poder usar los *Microsoft BingMaps SOAP Services* tenemos que agregar las referencias de servicio en nuestro proyecto. Para ello hacemos clic con el botón derecho en la carpeta llamada *Service Refence* que tenemos en el explorador de soluciones y seleccionamos “Agregar referencia de servicio...” (Ilustración 44):

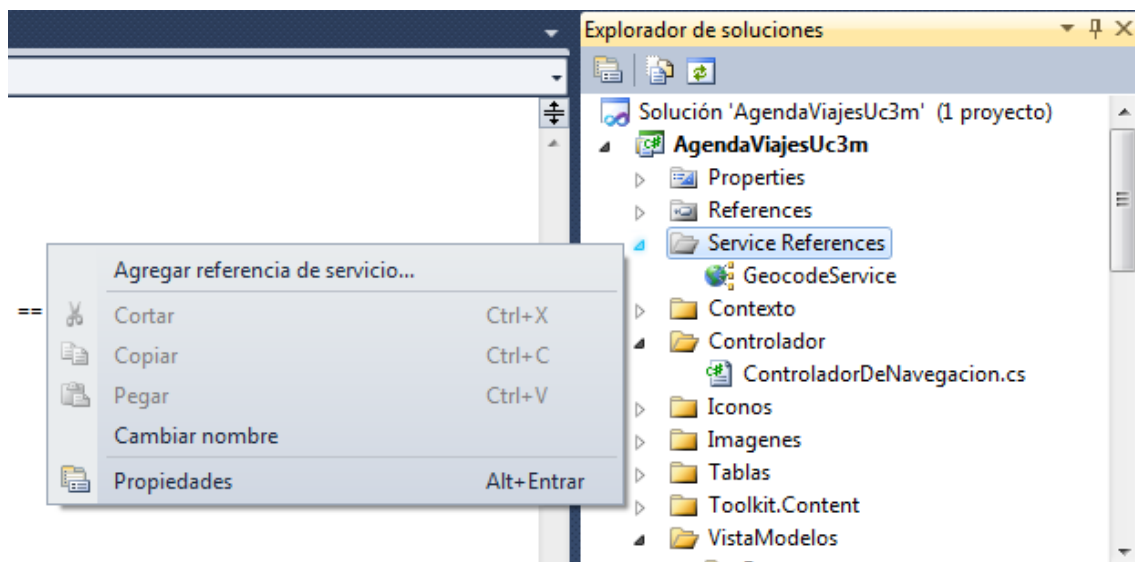


Ilustración 44. Agregar referencia de servicio.

En la ventana que nos salta a continuación ingresamos la URL (<http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc>) del servicio de geolocalización, que es el que vamos a usar, en el campo dirección y presionamos el botón “Ir” (Ilustración 45):

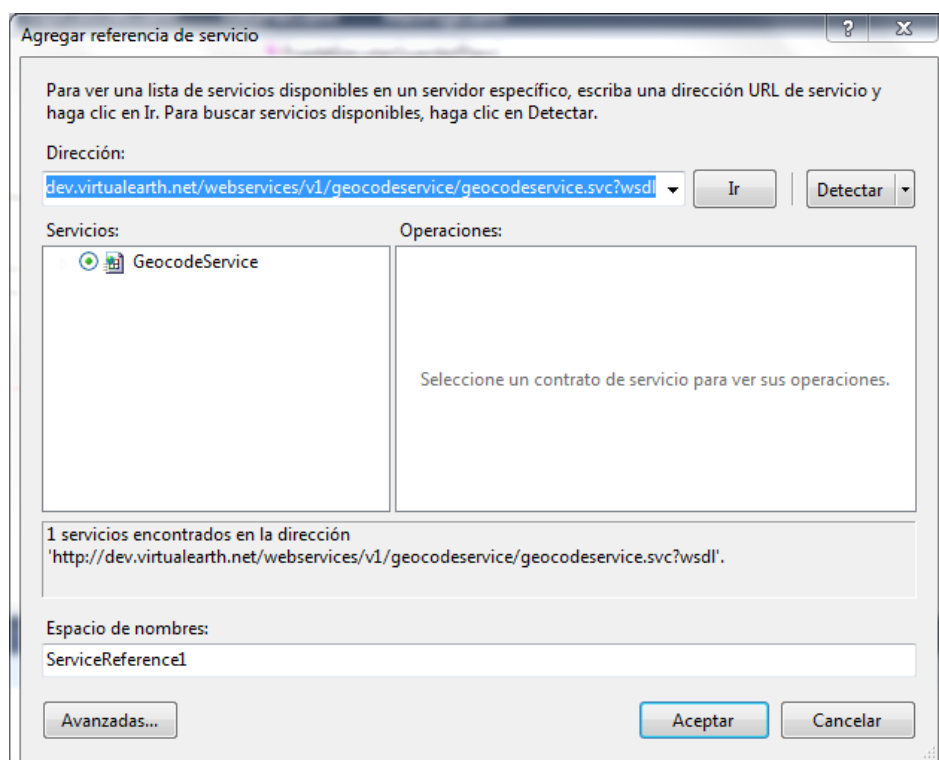


Ilustración 45. Configuración de una referencia de servicio.

Como vemos en la imagen, tras localizar y descargar el servicio, nos aparece este marcado en el campo “Servicios”. A continuación le damos un nombre al servicio y presionamos el botón “Avanzadas”. Esto nos muestra el siguiente diálogo (Ilustración 46):

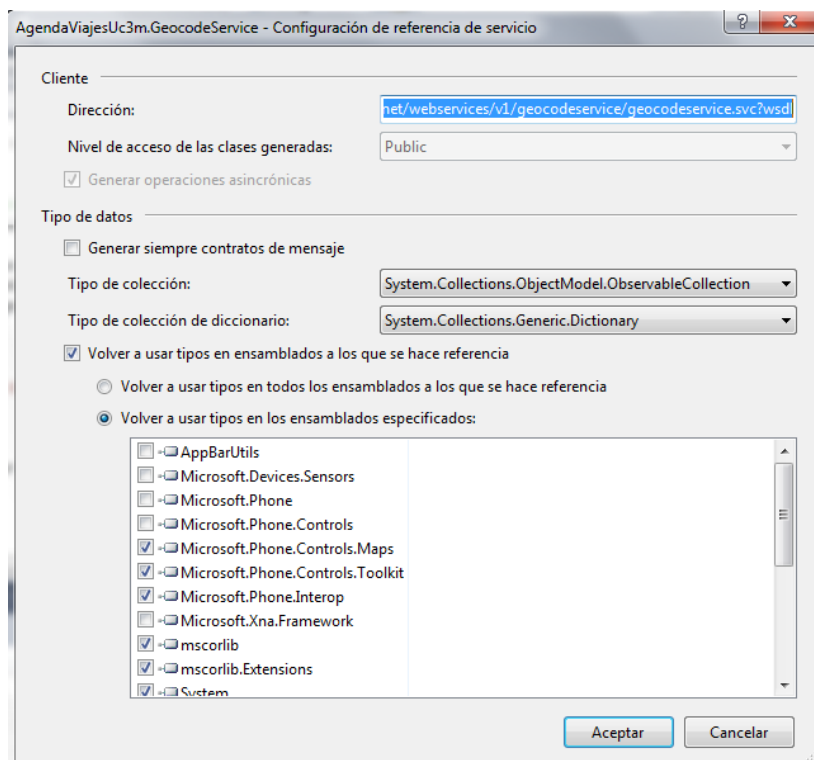


Ilustración 46. Configuración avanzada de una referencia de servicio.

Dentro de la configuración avanzada tenemos que cambiar el valor del “Tipo de colección” a *System.Collections.ObjectModel.ObservableCollection*. Dejando el valor por defecto nos da un error de incompatibilidades al compilar ya que en nuestro código esperamos que la respuesta nos la devuelva en este tipo de datos. Igual que en el “Tipo de colección de diccionario”, donde escogeremos la opción *System.Collections.Generic.Dictionary*.

También marcaremos la opción “Volver a usar tipos en los ensamblados específicos”. De esta manera, cuando se seleccionan, únicamente se volverán a usar los tipos seleccionados en la lista Ensamblados a los que se hace referencia.

Para finalizar pulsaremos el botón “Aceptar” y ya tendremos el servicio de geolocalización listo para ser usado. Vamos a ver a continuación cómo consultar esos servicios en el método *calcularCoordenadas()*:

```
private void calcularCoordenadas (string nombreViaje)
{
    GeocodeRequest geocodeRequest = new GeocodeRequest();
```

En este método nos creamos un objeto de tipo *GeocodeRequest*, que es la clase que contiene todas las propiedades necesarias para hacer una petición al servicio, y le añadimos los credenciales y la consulta que queremos hacer.

Para trabajar con el elemento *Bing Maps*, uno de los requisitos es obtener una clave de uso o credencial en el centro de cuentas de *Bing Maps* [8]. Para obtener dicha clave hemos tenido que iniciar sesión con una cuenta *LiveID* y especificado los datos de la nueva cuenta de *Bing Maps* (Ilustración 47):

Ilustración 47. Datos de nuestra nueva cuenta.

Tras presionar el botón *Save* podremos ir al menú de la izquierda, en el apartado *My Account* y seleccionar la opción *Create or View keys*. En este apartado es donde nos pedirán el nombre de nuestra aplicación, una URL opcional y el tipo de aplicación (Ilustración 48):

Ilustración 48. Registro de nuestra aplicación.

Al presionar el botón *Submit* se nos mostrará la clave que ha sido asignada para nuestra aplicación. Podemos tener hasta un máximo de 5 claves activas y consultar en cualquier momento nuestra clave, entrando al portal **www.bingmapsportal.com** con nuestro usuario y volviendo a la opción *Create or View keys*.

En caso de no obtener dichos credenciales, nos saldrá un aviso muy molesto en el mapa indicándonos que debemos solicitar una clave de uso del api de *Bing Maps* (Ilustración 49):



Ilustración 49. Falta de credenciales para el uso del elemento Bing Maps.

Una vez obtenidos los credenciales, los añadimos al *GeoCode Request*:

```
geocodeRequest.Credentials = new Credentials();
geocodeRequest.Credentials.ApplicationId =
"Ah0g_KrqMi1coz4FK7P_DZw_sZGiIMZxxUCRfpGgjtjmVoEsmhMZWEKZ0KicG_a";
```

A continuación asignamos la variable “nombreViaje”, que contiene un *String* extraído del TextBox de la interfaz, a la *Query*:

```
geocodeRequest.Query = nombreViaje;
```

El siguiente paso sería configurar los filtros. Creamos un *array* en el que solo incluimos un filtro que usamos para especificar un mínimo de confiabilidad en los resultados. En este filtro establecemos su propiedad *MinimumConfidence* con el valor *High*, de esta forma indicamos al servicio de geocodificación el nivel más alto de confianza en el resultado:

```
FilterBase[] filters = new FilterBase[1];
filters[0] = new ConfidenceFilter() { MinimumConfidence =
VisualCExp.GeocodeService.Confidence.High };
```

Añadimos ahora los filtros creados a las opciones de la petición:

```
GeocodeOptions geocodeOptions = new GeocodeOptions();
geocodeOptions.Filters = new ObservableCollection<FilterBase>(filters);
geocodeRequest.Options = geocodeOptions;
```

Ahora que la petición ha sido creada enviamos ésta al *Geocode ServiceClient*. En Silverlight es necesario crear un controlador de eventos que será lanzado en respuesta a la petición. Esto es similar a una devolución de llamada (vemos en el siguiente párrafo como se crea, *geocodeService_GeocodeCompleted*). Una vez lanzado el controlador de eventos, realizamos una llamada asíncrona al servicio de geocodificación:

```
if (nombreViaje != "")
{
    GeocodeServiceClient geocodeService = new
    GeocodeServiceClient("BasicHttpBinding_IGeocodeService");
    geocodeService.GeocodeCompleted += new
    EventHandler<GeocodeCompletedEventArgs>(geocodeService_GeocodeCompleted);
    geocodeService.GeocodeAsync(geocodeRequest);
}
else
{
    MessageBox.Show("Por favor, introduzca un término valido");
}
```

A continuación, creamos un método que se activa cuando el servicio de geocodificación responde. Un objeto *GeocodeResponse* es devuelto por el método del servicio web. Tenemos que extraer el resultado de este método y mostrar sus coordenadas:

```
void geocodeService_GeocodeCompleted(object sender, GeocodeCompletedEventArgs e)
{
    GeocodeResponse geocodeResponse = e.Result;
    GeoCoordinate coordenadas = new GeoCoordinate();
    if (geocodeResponse.Results.Count > 0)
    {
        coordenadas.Latitude =
        geocodeResponse.Results[0].Locations[0].Latitude;
        coordenadas.Longitude =
        geocodeResponse.Results[0].Locations[0].Longitude;
    }
    else
    {
        MessageBox.Show("El nombre del viaje no corresponde a ningún lugar");
        coordenadas.Latitude = 40.19;
        coordenadas.Longitude = -3.46;
    }
    Coordenadas = coordenadas;
}
```

Como se ve en el código, en caso de que la respuesta venga vacía rellenamos las coordenadas con los datos de latitud y longitud de Leganés.

vi. Aumentar y disminuir el zoom

El código de los botones que controlarán el zoom del mapa es muy sencillo, para ello nos creamos dos nuevos comandos con los que controlar el nivel de zoom que se puede alcanzar:

```
public bool PuedeEjecutarMasZoom()
{
    bool dev = true;
    if (ZoomLevel == 18) { dev = false; };
    return dev;
}

public void EjecutarMasZoom()
{
    ZoomLevel += 1;
}
```

Como vemos en el código, el botón permanecerá activo hasta que el nivel de zoom alcance el máximo, que está establecido en 18. De esta forma cada vez que el usuario presione el botón “+” del zoom aumentará el nivel en una unidad. El código para alejar el zoom será prácticamente igual, solamente tendremos que ir restando unidades al nivel de zoom, en vez de sumarle.

e) Detalle de los días

La última pantalla que queda por ver en la parte de planificación de nuestra aplicación es la que permite al usuario introducir una descripción sobre lo que tiene pensado hacer durante el viaje en el día seleccionado. Por tanto, el usuario se tiene que situar en la primera vista del elemento *Pivot* de la página *EditaPlan.xaml* y tras seleccionar uno de los días, accederá a la página *Descripcion.xaml* (Ilustración 50):



Ilustración 50. Página *EditaPlan.xaml* y *Descripcion.xaml*

Esta página está dividida en tres elementos. En la parte superior de la pantalla tenemos un *TextBlock* que se encarga de informar al usuario del día que está modificando. En medio y ocupando la mayor parte de la superficie del *Frame* tenemos un elemento *TextBox* donde el usuario podrá escribir sus planes para ese día. Por último, una barra de aplicación con las opciones de confirmar y cancelar. Todos estos elementos están conectados con la vista modelo de la página mediante enlace a datos.

En el *TextBox* de esta página cabe señalar dos propiedades que hemos modificado para mejorar la usabilidad. Se trata de la propiedad *TextWrapping*, la cual puede tomar los valores *Wrap* y *NoWrap*. Nosotros hemos escogido el valor *Wrap*. Con este valor, cuando el texto alcanza el borde del *TextBox* salta de línea, mientras que si el valor fuese *NoWrap* el *TextBox* se expandiría ocupando todo el espacio que requiriese el texto.

La otra propiedad a reseñar es *AcceptsReturn*, la cual establecemos a *true*. Con esta propiedad marcada, cada vez que el usuario pulse el botón “Salto de línea” del teclado el cursor saltará de línea.

En esta ocasión en el *code behind* solamente hemos introducido código para inicializar los elementos y rescatar el identificador del día en el que queremos introducir alguna descripción.

Como hemos dicho anteriormente, todos los elementos están enlazados con los datos, incluido el *TextBox*. En este punto hubo que solventar un problema, ya que en la base de datos el número de día que ocupa un día en el viaje está guardado como un entero, y el *TextBox* quedaría poco claro si solo se le mostrase al usuario un número, por lo que se ha modificado el dato añadiendo la palabra “Día” antes de ser mostrado:

```
public VMDescripcion(int parametro)
{
    this.parametro = parametro;
    using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
    {
        Listadias = (from Dia dia in contextDb.Dias where (dia.DiaId ==
parametro) select dia).ToList();
        BoxDescripcion = Listadias.ElementAt(0).Descripcion;
        int numDia = Listadias.ElementAt(0).NumDia;
        BlockDia = "Dia " + numDia;
    }
}
```

En cuanto al botón “Guardar” de la barra de aplicaciones, no incluye novedades respecto a los vistos anteriormente. El usuario podrá introducir texto y en cualquier momento presionar el botón de “Guardar”, para que en ese momento, se lancen los comandos de enlace a datos que permiten leer el texto introducido en el *TextBox*, subirlo a la base de datos y abandonar la página.

XI. Modo “Durante el viaje”

Una vez que se ha analizado el funcionamiento de la parte de planificación de la aplicación, es momento de ver la parte en la que se pretende que el usuario utilice dicha aplicación para el transcurso de sus viajes. Para acceder a este modo el usuario tiene que pulsar el botón “Durante el viaje” de la página principal (Ilustración 51):



Ilustración 51. Página MainPage.xaml y Viajes.xaml

En esta parte de la aplicación, cuando el usuario seleccione un viaje podrá añadir los gastos que le vayan surgiendo durante el transcurso del mismo. Además tendrá un par de herramientas para no perderse: una brújula digital y una calculadora de rutas con indicaciones entre dos puntos.

La primera pantalla a la que accedemos es Viajes.xaml, en la cual tenemos un elemento *ListBox* conectado con la vista modelo de la página mediante la propiedad *ItemSource*. Esta propiedad es la encargada de surtir de elementos a la lista que vamos a mostrar, que en este caso listará los viajes creados anteriormente por el usuario. Mediante la propiedad *DisplayMemberPath* indicamos el campo que queremos mostrar en la lista de cada elemento Viaje:

```
<ListBox Name="listaViajes"
    DisplayMemberPath="Nombre" Style="{StaticResource estiloListBox}"
    SelectedItem="{Binding ItemSeleccionado, Mode=TwoWay}"
    SelectionChanged="lstViajes_CambioSeleccion"
    ItemsSource="{Binding ListaViajes, Mode=TwoWay}">
</ListBox>
```

El uso de esta pantalla será simplemente elegir el viaje que se quiera para que la aplicación navegue hacia una página específica de ese viaje. Para ello utilizamos la propiedad del *ListBox* *SelectedItem* combinado con el evento *SelectionChanged*.

El evento *SelecionChanged* entra en funcionamiento cuando el usuario selecciona un ítem de la lista y dentro del evento llamamos al comando de la vista modelo *CambioSeleccionado*:

```
private void lstViajes_CambioSeleccion(object sender, SelectionChangedEventArgs e)
{
    contexto.CambioSeleccion.Execute("{Binding CambioSeleccion}");
}
```

Dentro del método *EjecutarCambioSeleccion()* lo primero que hay que hacer es comprobar que el ítem seleccionado es distinto de nulo (*null*). Como bien explicamos anteriormente, esta comprobación hay que hacerla porque tras volver a esta página, el método se ejecuta por sí solo, ya que interpreta que ha habido cambios en la selección. El ítem seleccionado es distinto al que había porque al volver se deselecta el ítem y esto el sistema lo considera un cambio, ahora el ítem seleccionado pasaría a ser nulo:

```
public void EjecutarCambioSeleccion()
{
    if (ItemSeleccionado != null) {
```

A continuación extraemos el identificador del ítem seleccionado, que como será un objeto de tipo “Viaje”, ya sabemos cómo hacerlo (capítulo 4.IX.d). Una vez extraído es momento de incluirlo como parámetro en la llamada a la siguiente página:

```
Int idViajeSel = ItemSeleccionado.ViajeId;
string parameters = string.Format("IDDIA={0}", idViajeSel);
controlador.ControladorDeNavegacion.Current.NavigateTo("ViajeConcreto", parameters);
```

XII. Página con las utilidades durante el viaje: cámara, ListPicker, múltiples AppBar, ListBox, ExpanderView, brújula

En el momento en el que el usuario ha seleccionado un viaje navega a una página en la que podrá interactuar con la información de su viaje en tiempo real. Es decir, está pensada para que sea usada mientras el usuario esté inmerso en su viaje.

Para que el uso de las utilidades sea más sencillo, hemos vuelto a optar por crear un página con diseño *Pivot*. De esta manera el usuario no tendrá que navegar entre páginas hacia delante y atrás, que en ciertas ocasiones resulta molesto. Por tanto la página *ViajeConcreto.xaml* estará dividida en tres vistas de tipo

Pivot. Una tendrá información sobre el presupuesto y los gastos, otra en la que se mostrará una lista con los gastos creados y la última que servirá de ayuda en la localización.

Además incluiremos un acceso directo a la cámara del sistema. La idea que se tenía en un principio era la de guardar las fotografías tomadas en una carpeta específica para cada viaje, de forma que luego, cuando el usuario accediese a la galería, las tuviese organizadas de manera más clara. El problema que nos encontramos aquí es que Microsoft ha pensado mucho en la seguridad de su sistema operativo, de manera que toda aplicación que se ejecuta en Windows Phone está aislada del sistema y del resto de aplicaciones. Esto implica que ninguna aplicación ajena al sistema operativo puede acceder a zonas de memoria usadas por el sistema o por otras aplicaciones. También impide, como es en nuestro caso, acceder a recursos generales del sistema, como por ejemplo, el directorio de almacenamiento. Tan solo disponemos de un almacenamiento “especial”, el cual sí puede ser compartido y accesible por cualquier aplicación: el MediaLibrary. Para acceder a él debemos hacer uso de los métodos de la clase MediaLibrary, la cual proporciona acceso a las canciones, listas de reproducción e imágenes de la biblioteca multimedia del dispositivo. Se tomó la decisión de que, una vez hecha la foto, el propio sistema volviese inmediatamente a nuestra aplicación.

- a. Interfaz de la vista gastos.
- b. Construcción de los elementos visuales de la vista gastos.
- c. Vista modelo de la primera vista del *Pivot*: gastos.
- d. Interfaz de la vista lista de gastos.
- e. Construcción de los elementos visuales de la vista lista de gastos.
- f. Vista modelo de la segunda vista del *Pivot*: lista de gastos.
- g. Interfaz de la vista navegación.
- h. Construcción de los elementos visuales de la vista navegación.
- i. Vista modelo de la tercera vista del *Pivot*: navegación.

a) Interfaz de la vista Gastos

Vamos a empezar hablando de la primera vista, que es la que se le muestra al usuario al seleccionar un viaje. Tenemos una pantalla en la que se informa de la cantidad que nos queda hasta llegar al máximo del presupuesto establecido y también podremos añadir gastos que surjan durante el viaje.

En esta vista vamos a tener dos partes diferenciadas. En la parte superior de la pantalla se informa al usuario del presupuesto máximo que introdujo en la planificación y a continuación, se le informa de los euros que le quedan para llegar a ese máximo. En la parte media baja de la pantalla se le presenta un pequeño formulario para añadir nuevos gastos.

Este formulario consta de tres campos. En el primero, que tendrá la novedad de introducir un elemento nuevo llamado *ListPicker*, deberá seleccionar el día del viaje en el que se ha producido el gasto. En el segundo campo deberá introducir un importe y en el tercero una descripción o el concepto por el que se ha producido el gasto. Una vez rellenados correctamente podrá pulsar el botón “Añadir”, situado en la barra de aplicación de la parte inferior de la pantalla. Automáticamente se calculará el dinero restante para llegar al máximo, actualizando así la información en tiempo real.

Por último, tenemos una barra de aplicación que contará con dos botones, uno que nos permita guardar la información con la que hayamos completado el formulario y otro que sirve como acceso directo a la cámara del sistema (Ilustración 52):



Ilustración 52. Vista gastos.

b) Construcción de los elementos visuales de la vista gastos

Para poder mostrarle la información al usuario relativa al viaje seleccionado, lo primero que hacemos es llamar al constructor de la página, y a continuación rellenar los diferentes campos con la información necesaria. Para ello sobrescribimos el método *OnNavigatedTo(NavigationEventArgs e)* pasándole el evento de navegación:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (newInstance)
    {
        if (contexto == null)
        {
            if (State.Count > 0)
            {
                contexto = (VMViajeConcreto)State["VMViajeConcreto"];
                contexto.mostrarGastos();
            }
            else
            {
                string aux = NavigationContext.QueryString["ID"];
                id = Convert.ToInt16(aux);
                contexto = new VMViajeConcreto(id);
                listPicker1.SelectedIndex = -1;
            }
        }
        this.DataContext = contexto;
    }
    newInstance = false;
}
```

Como en otras vistas modelos primero hacemos las comprobaciones para saber si nos encontramos con una página que se está iniciando o que “despierta” después de haber salido de la aplicación. En caso de

que tengamos que recuperarnos de un estado “durmiente”, restablecemos la información guardada en la variable *State* y llamamos al método de la vista modelo *mostrarGastos()*.

Si estamos ante una instancia nueva de la clase, rescatamos el dato que nos pasan como parámetro en la llamada a la página. Este dato se lo pasamos al constructor de la vista modelo, ya que se trata del identificador de viaje. Gracias a él, vamos a saber qué viaje ha seleccionado el usuario. También tenemos que inicializar el elemento *ListPicker* a su valor por defecto, ya que si no hacemos esto, se produce un error en la lectura de los datos. Por último, asignamos la vista modelo llamada “contexto” al *DataContext*.

Para terminar de ver cómo inicializar todos los componentes tenemos que ver el constructor de la clase *ViajeConcreto.xaml.cs*. En él vamos a inicializar la barra de aplicación de la parte inferior. En esta ocasión hemos tenido que crear la barra de aplicación por código ya que en la presentación no se está permitida la creación de varias barras de aplicación para una misma página. En esta página vamos a necesitar dos barras de aplicación distintas, una para la primera vista del *Pivot* y otra para la segunda:

```
AppBar Uno = new AppBar();
AppBar Dos = new AppBar();
```

Lo primero que hacemos en el código es establecer la barra de aplicación que queremos mostrar y establecer sus propiedades *IsVisible* e *IsMenuEnable* a *true* para que la barra de aplicación sea visible y permita habilitar el menú oculto respectivamente:

```
AppBar = AppBarUno;
AppBar.IsVisible = true;
AppBar.IsMenuEnabled = true;
```

A continuación nos creamos los botones que vamos a necesitar en la barra de aplicación. Para la primera vista nos crearemos los botones que permiten añadir gastos y abrir la cámara de fotos del sistema:

```
AppBarIconButton añadir = new AppBarIconButton(new
Uri("/Iconos/appbar.save.rest.png", UriKind.Relative));
AppBarIconButton irCamara = new AppBarIconButton(new
Uri("/Iconos/appbar.feature.camera.rest.png", UriKind.Relative));

añadir.Text = "Añadir gasto";
irCamara.Text = "Hacer una foto";
AppBarUno.Buttons.Add(añadir);
AppBarUno.Buttons.Add(irCamara);

añadir.Click += new EventHandler(btnAñadir);
irCamara.Click += new EventHandler(btnCamara);
```

Por último recogemos el evento *Click* de los botones, lanzando los métodos *btnAñadir()* y *btnCamara()* respectivamente:

```
void btnAñadir(object sender, EventArgs e)
{
    contexto.ComandoAñadirGasto.Execute("Binding ComandoAñadirGasto");
    this.Focus();
}
```

La funcionalidad del botón “Añadir” simplemente es la de llamar al comando que ejecuta el método *AñadirGasto()* y liberar del foco al *TextBox* correspondiente al contexto. Si no hubiésemos añadido esa última línea al presionar el botón “Añadir” los cambios se guardarían correctamente pero el teclado correspondiente al *TextBox* quedaría visible produciéndose una sensación extraña en la percepción de los cambios realizados.

Gracias a que en esta aplicación conservamos el estado, al volver a ella el usuario se encuentra la pantalla donde estaba tal y como la dejó. Por este motivo, desde dentro de la aplicación podemos comunicarnos con el entorno que la rodea, es decir, el sistema operativo.

Para poder realizar acciones que activen parte del sistema, como puede ser realizar una llamada de teléfono, enviar un mensaje o sacar una foto, Windows Phone introduce el concepto de lanzador y selector.

Un selector es una API que tenemos a nuestra disposición para usar partes del sistema con la finalidad de obtener datos, esto es: sacar una foto, elegir una imagen de la galería de imágenes o seleccionar una dirección de correo electrónico de la libreta de contactos, entre muchos más.

Al ejecutar un selector, al igual que ocurre con los lanzadores, salimos de nuestra aplicación, por lo que entra en funcionamiento el ciclo de *tombstoning*. La mayor diferencia con los lanzadores es que, en el caso de los selectores, esperamos una respuesta por parte del selector que hemos iniciado.

Para poder abrir la aplicación de la cámara de nuestro dispositivo, es necesario añadir un *using* a *Microsoft.Phone.Tasks* en el archivo *ViajeConcreto.xaml.cs*:

```
using Microsoft.Phone.Tasks;
```

Y modificar el código subyacente al botón que da acceso a la cámara para crear una nueva instancia de *CameraCaptureTask*:

```
void btnCamara(object sender, EventArgs e)
{
    CameraCaptureTask camCapture = new CameraCaptureTask();
    camCapture.Show();
}
```

De esta manera podremos acceder rápidamente desde nuestra aplicación a la cámara de fotos del sistema operativo (Ilustración 53):

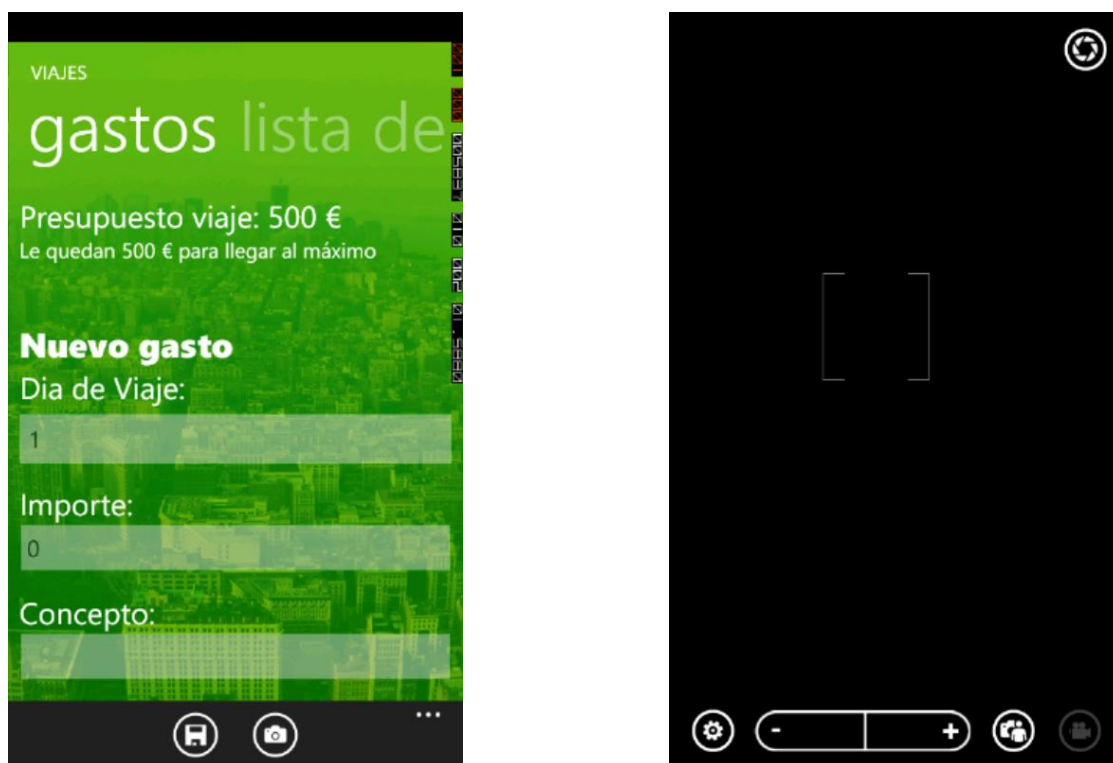


Ilustración 53. Acceso a la cámara del sistema.

Tras hacer una foto, la cual se almacenará en la memoria interna del dispositivo, el sistema volverá automáticamente a nuestra aplicación.

c) Vista modelo de la primera vista del Pivot: Gastos

Vamos a empezar viendo el código de la vista modelo que hace funcionar la parte superior de la vista, aquella en la que se da la información relativa al presupuesto (Ilustración 53):



Ilustración 54. Vista gastos.

Como vemos en la imagen, cuando accedemos a la vista gastos y no tenemos añadido ningún gasto, la primera frase nos informa del presupuesto máximo del viaje. Esta cifra solo se podrá modificar en la parte de planificación de viaje. En la segunda frase se nos informa de la cantidad que nos queda hasta llegar al máximo establecido. Esta cifra irá disminuyendo según vayamos introduciendo gastos (Ilustración 54).

Llegará un momento en que estemos cerca de llegar a ese máximo, para tal caso hemos decidido avisar al usuario cambiando el color de la cifra a rojo. Ese momento lo hemos fijado en el 80% del máximo. Es decir, cuando el gasto total supere el 80% del presupuesto máximo la cifra pasará a mostrarse en color rojo (Ilustración 55).

Y por último si el usuario sigue añadiendo gastos, de forma que el total supera el máximo que se marcó en un principio, cambiamos la frase para indicar la cifra del importe en que ha superado ese máximo, además de mantener la cifra en rojo (Ilustración 56).

Para hacer posibles todas estas comprobaciones tenemos que ver la vista modelo de la página ViajeConcreto.xaml. En el constructor de la VMViajeConcreto.cs lo primero que hemos hecho es inicializar todas las variables que vamos a controlar de la interfaz:

```
texto1 = "Le quedan ";  
texto2 = " € para llegar al máximo";  
gastoTotal = 0;
```



Ilustración 55. Vista gastos después de haber insertado varios gastos.



Ilustración 56. Vista gastos habiendo superado el 80% del presupuesto máximo.



Ilustración 57. Vista gastos habiendo superado el presupuesto máximo.

A continuación abrimos la conexión con la base de datos para poder seleccionar el viaje con el que se quiere trabajar, para ello utilizamos el parámetro que nos pasaron en la navegación:

```
public VMViajeConcreto(int parametro)
{
    this.parametro = parametro;
    using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
    {
        listaViajes = (from Viaje viaje in contextDb.Viajes where viaje.ViajeId
== parametro select viaje).ToList();
    }
}
```

Nada más extraer el viaje en concreto accedemos al presupuesto para mostrárselo al usuario en pantalla:

```
presupuesto = listaViajes.ElementAt(0).Presupuesto;
```

Esta variable llamada “presupuesto” está conectada mediante la técnica de enlace a datos con la propiedad de texto de un elemento *TextBlock* de la interfaz. Para obtener toda la funcionalidad debemos implementar los métodos *get()* y *set()*:

```
public int Presupuesto
{
    get
    {
        return presupuesto;
    }
    set
    {
        presupuesto = value;
    }
}
```

Una vez extraído el presupuesto máximo del viaje accedemos a la tabla “Gastos” para crearnos una lista de elementos “Gastos” que contendrá todos los gastos pertenecientes al viaje, es decir aquellos cuyo “IdViaje” coincida con el parámetro que nos han pasado:

```
listaGastos = (from Gasto gasto in contextDb.Gastos where gasto.ViajeId == parametro
select gasto).ToList();
```

A continuación, se recorre la lista de gastos, y de cada gasto accedemos a su importe. De esta forma, los vamos sumando para calcular el gasto total acumulado de cada viaje. Posteriormente, calcularemos la diferencia entre el presupuesto y el gasto total, para mostrarle el resultado al usuario mediante la misma técnica usada con la variable “Presupuesto”:

```
private int gastoTotal = 0;
for (int i = 0; i < listaGastos.Count; i++)
{
    gastoTotal = gastoTotal + listaGastos.ElementAt(i).Importe
}
DifGastoTotal = Presupuesto - gastoTotal;
```

Como se puede dar el caso de que el gasto total sea mayor que el presupuesto máximo, habrá que controlar que la variable “DifGastoTotal” no se muestre en negativo, para resolver este pequeño detalle simplemente calculamos el valor absoluto de esa variable:

```
ValorAbsolutoMostrado = Math.Abs(DifGastoTotal);
```

Para comprobar si la cifra la tenemos que mostrar en rojo o no, tenemos que calcular el 80% del máximo, lo vemos en las siguientes líneas de código:

```
LimiteRojo = calcularPorcentaje(Presupuesto);
if (DifGastoTotal < LimiteRojo)
{
    if (DifGastoTotal < 0)
    {
        Texto1 = "Ha sobrepasado ";
        Texto2 = " € el presupuesto máximo";
    }
    colorGasto = "Red";
} else { colorGasto = "White"; }
```

En la variable “LimiteRojo”, guardamos el valor que representa el 80% del presupuesto máximo. Una vez calculada esa cifra la comparamos con la diferencia del gasto total con el presupuesto máximo. Si no se cumple la condición, el color de la cantidad mostrada será blanco, pero en caso de que se cumpla la condición, la cifra se mostrará en rojo. Tan solo falta ver si además tenemos que cambiar la frase porque el gasto total ha superado al presupuesto máximo.

Al final, el elemento “TextBlock” que tendremos en la página XAML quedará así:

```
<StackPanel Grid.Row="1" Orientation="Horizontal">
    <TextBlock Text="{Binding Texto1, Mode=TwoWay}" Style="{StaticResource
estiloBlock}" FontSize="22"></TextBlock>
    <TextBlock Name="textQueda" Text="{Binding ValorAbsolutoMostrado,
Mode=TwoWay}" Style="{StaticResource estiloBlock}" Foreground="{Binding
ColorGasto, Mode=TwoWay}" FontSize="22"></TextBlock>
    <TextBlock Text="{Binding Texto2, Mode=TwoWay}" Style="{StaticResource
estiloBlock}" FontSize="22" Margin="0,0,0,60"></TextBlock>
</StackPanel>
```

Como vimos anteriormente, tenemos que calcular el 80% del presupuesto máximo. Para realizar esa tarea hemos creado un método al que le pasamos una cantidad como parámetro y nos devuelve el 80% de esa cantidad:

```
public int calcularPorcentaje(int presupuestoMax)
{
    int aux = presupuestoMax * 80;
    aux = aux / 100;
    int ochenta = presupuestoMax - aux;
    return ochenta;
}
```

Una vez vista toda la operativa de la parte superior de la vista Gastos, vamos a ver ahora la parte inferior, en la que se le presenta al usuario un formulario de tres campos para añadir nuevos gastos. En ella cabe destacar el primer campo. Se ha optado por incluir una herramienta nueva proporcionada por el paquete extra *Toolkit*. Se trata de la herramienta *ListPicker*. Con este elemento podemos mostrar listas desplegables de cualquier tipo de elementos:

```
<toolkit:ListPicker Name="listPicker1" Grid.Row="1"
    FullModeHeader="Dias del Viaje"
    ItemsSource="{Binding ListaDesplegable}" Opacity="0.5"
    SelectedIndex="{Binding DiaSeleccionado, Mode=TwoWay}"
    Background="LightGray" Margin="0" />
```

El control *ListPicker* lo tenemos disponible dentro del Windows Phone Silverlight Toolkit que ya vimos anteriormente cómo instalarlo (ver capítulo 3.V.c). Este control nos permite mostrar una serie de opciones a elegir por parte del usuario (de manera muy similar a como lo haríamos utilizando un *ComboBox*). Deriva de la clase *ItemsControl* y permite una gran personalización. El uso de *data bindings* nos facilita un gran control sobre la interacción que el usuario realiza con él mediante el uso de eventos. Tiene dos tipos de selecciones dependiendo de la cantidad de elementos que contenga:

- Cinco o menos: Expandida en el mismo lugar donde se define (como lo haría un *ComboBox*).
- Más de cinco: En un *popup* a pantalla completa (muy útil si se va a mostrar un número elevado de elementos) (Ilustración 57).

Para cargar el *ListPicker* con los días de que consta el viaje, primero tenemos que acceder a la tabla “Dias” y seleccionar aquellos cuyo “IdViaje” corresponda con el parámetro que nos pasaron:

```
listaDesplegable = (from Dia dia in contextDb.Dias where dia.ViajeId == parametro
select dia.NumDia).ToList();
```

Como en la pantalla tenemos un formulario con tres campos, una vez completados, el usuario podrá pulsar el botón que permite guardar el gasto. Vamos a ver a continuación cómo es el código que se ejecuta al pulsar el botón “Añadir”, que se encuentra en la barra de aplicación:

```
<shell:AppBarIconButton IconUri="\Icons\appbar.add.rest.png" Text="Añadir
gasto" />
<au:AppBarItemCommand Id="Añadir gasto" Command="{Binding ComandoAñadirGasto,
Mode=TwoWay}"/>
```

Al igual que para el resto de la aplicación, hacemos uso de los comandos para enlazar los botones con la vista modelo. Como en todos los comandos, tenemos dos métodos: uno en el que comprobamos si se puede ejecutar y otro en el que está el código de lo que se pretende hacer en ese comando:


```

public DelegateCommand ComandoAñadirGasto
{
    get
    {
        if (comandoAñadirGasto == null)
            comandoAñadirGasto = new DelegateCommand(EjecutarAñadirGasto,
                                                         PuedeEjecutarAñadirGasto);

        return comandoAñadirGasto;
    }
}

```

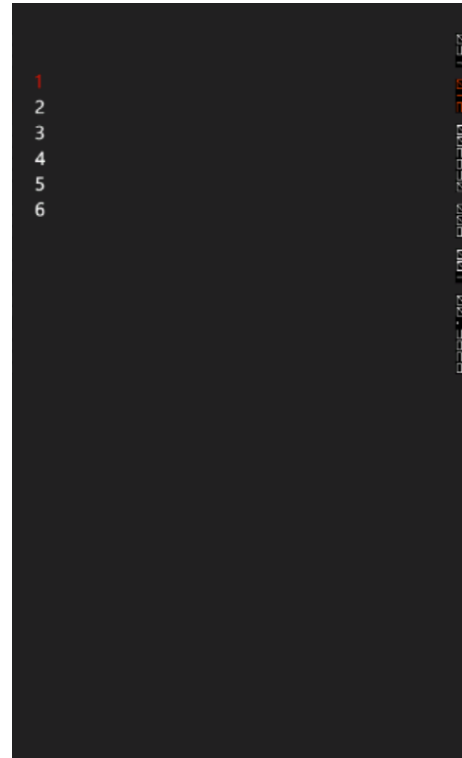


Ilustración 58. *ListPicker* en modo expandido y en pantalla completa.

El botón “Añadir” siempre va a estar activo, por lo que *PuedeEjecutarAñadirGasto()* siempre devolverá *true*. Las comprobaciones de que los datos del formulario se han introducido correctamente las haremos directamente en el método *EjecutarAñadirGasto()*:

```

public bool PuedeEjecutarAñadirGasto()
{
    return true;
}

```

En el método *EjecutarAñadirGasto()*, lo primero que haremos para mejorar la fluidez es quitarle el foco de la acción. De esta manera, aunque el usuario pulse el botón “Añadir” estando el teclado activo, los valores introducidos serán correctamente procesados:

```

public void EjecutarAñadirGasto()
{
    object focusObj = FocusManager.GetFocusedElement();
    if (focusObj != null && focusObj is TextBox)
    {
        var binding = (focusObj as
            TextBox).GetBindingExpression(TextBox.TextProperty);
        binding.UpdateSource();
    }
}

```

A continuación comprobamos que el formulario está correctamente relleno:

```
if (importe < 1 || concepto == string.Empty)
{
    MessageBoxResult result = MessageBox.Show("Debe indicar el concepto y el importe antes de añadir un gasto", "Error", MessageBoxButton.OK);
}
```

Si el usuario no ha introducido ningún importe (en cuyo caso el valor de importe sería 0) o no ha rellenado el campo concepto, no se le permite añadir ningún gasto, por lo que se le muestra un mensaje informativo.

Tenemos que contemplar la posibilidad de que el usuario añada un gasto el primer día de su viaje. Para eso es posible que no toque el elemento *ListPicker* (ya que este muestra el uno por defecto) y por tanto la variable “DiaSeleccionado” venga con valor *null*. En tal caso, buscaremos en la base de datos el identificador del día seleccionado usando la variable “enteroAux”, cuyo valor será de 1.

En caso de que el usuario interactúe con el *ListPicker* y seleccione él mismo el día en el que añadir el gasto, tendremos que convertir el dato en entero y guardarlo en la variable “enteroAux”, añadiéndole uno. Esto lo hacemos porque el objeto que obtenemos del *ListPicker* es un *ListPickerElement* que contiene el índice del elemento seleccionado. Por ejemplo si seleccionamos el día 2, el índice de ese día será 1:

```
else
{
    List<int> listaAux;
    int enteroAux = 1;
    if (diaSeleccionado != null)
    {
        string cadenaAux = diaSeleccionado.ToString();
        enteroAux = Convert.ToInt16(cadenaAux);
        enteroAux = enteroAux + 1;
    }
}
```

A continuación, abrimos la conexión con la base de datos y hacemos un *Select* para obtener el identificador del día en el que queremos añadir el gasto. Para hacer la búsqueda tenemos los campos “ViajeId” y “numDia” para que la consulta nos devuelva de manera inequívoca dicho identificador:

```
using (AppDbContext context = new AppDbContext("Data Source='isostore:/MiViajeDb.sdf'"))
{
    listaAux = (from Dia dia in context.Dias where dia.ViajeId == parametro && dia.NumDia == enteroAux select dia.DiaId).ToList();
}
```

El siguiente paso será crear una lista de elementos tipo “Gasto”. En esta lista crearemos el nuevo gasto que queremos añadir en la base de datos:

```
List<Gasto> nuevoGasto = new List<Gasto>();
nuevoGasto.Add(new Gasto()
{
    ViajeId = listaViajes.ElementAt(0).ViajeId,
    DiaId = listaAux.ElementAt(0),
    Importe = importe,
    Concepto = concepto
});
```

Los campos “ViajeId” y “DiaId” los obtenemos de las búsquedas que hicimos anteriormente. Mientras que los campos “Importe” y “Concepto” directamente los recogemos de la interfaz.

A continuación añadimos la lista con el gasto creado en un estado de pendiente inserción, para posteriormente enviar los cambios:

```
context.Gastos.InsertAllOnSubmit(nuevoGasto);
context.SubmitChanges();
```

Al pulsar el botón “Añadir Gasto” hemos conseguido insertar un nuevo gasto en la base de datos, pero también queremos que el texto que informa al usuario sobre el dinero que aún le queda para llegar al máximo, se actualice. Por lo que después de subir los cambios a la base de datos hemos recalculado esa cantidad:

```
gastoTotal = gastoTotal + importe;
DifGastoTotal = Presupuesto - gastoTotal;
ValorAbsolutoMostrado = Math.Abs(DifGastoTotal);
```

Primero calculamos el gasto total, que saldrá de la suma del gasto total que teníamos antes de añadir el gasto, más el importe del gasto. A continuación, le restamos al presupuesto total del viaje, el gasto total calculado anteriormente, y ya tenemos en la variable “DifGastoTotal” la cantidad que queremos mostrar al usuario. Por último calculamos el valor absoluto para mostrar siempre valores positivos en la interfaz.

Una vez guardados los datos del nuevo gasto, es momento de resetear los campos de la interfaz e informar al usuario de que el gasto ha sido añadido:

```
Importe = 0;
Concepto = string.Empty;
MessageBox.Show("El gasto ha sido añadido correctamente");
```

Finalmente, antes de salir del método llamamos a *mostrarGastos()*. En este método que veremos en detalle más adelante, haremos, entre otras cosas, los cálculos necesarios para mostrar la cifra que informa del importe que resta para llegar al máximo, en blanco o en rojo, según explicamos anteriormente.

d) Interfaz de la vista Lista de gastos: ListBox

La segunda vista del *Pivot* está dedicada a mostrar una lista de los gastos que el usuario va añadiendo. Además, se le ofrecerá la posibilidad de eliminar el gasto que crea conveniente pulsando uno de los botones de la barra de aplicación. El otro botón que encontrará le servirá, como en la vista anterior, para acceder directamente a la cámara de fotos del sistema (Ilustración 58).

Como vemos en el código XAML de la página, la lista de gastos la vamos a mostrar mediante el elemento *ListBox*:

```
<ListBox Name="listaViajes" Style="{StaticResource estiloListBox}"
          ItemsSource="{Binding ListaDEexpander, Mode=TwoWay}"
          ItemContainerStyle="{StaticResource estiloGastos}">
</ListBox>
```

a) Construcción de los elementos visuales de la vista lista de gastos.

Anteriormente, vimos cómo nos creábamos la barra de aplicación por código, ya que en esta página vamos usar dos barras de aplicaciones, una para la vista gastos y otra para la vista que nos ocupa. Para esta vista nos vamos a crear la barra de aplicación llamada “AppBarDos” y en ella insertaremos dos botones: un botón para eliminar gastos y otro botón (que será el mismo que el usado en la vista anterior) para acceder a la cámara del sistema:

```

AppBar AppBarDos = new AppBar();

AppBarIconButton borrar = new AppBarIconButton(new
Uri("/Iconos/appbar.delete.rest.png", UriKind.Relative));
AppBarIconButton irCamara = new AppBarIconButton(new
Uri("/Iconos/appbar.feature.camera.rest.png", UriKind.Relative));

borrar.Text = "Eliminar gasto";
irCamara.Text = "Hacer una foto";

AppBarDos.Buttons.Add(borrar);
AppBarDos.Buttons.Add(irCamara);

irCamara.Click += new EventHandler(btnCamara);
borrar.Click += new EventHandler(btnBorrar);

```



Ilustración 59. Lista de gastos antes y después de agregar elementos.

Como para la vista anterior ya explicamos detalladamente el código que crea las barras de aplicaciones, vamos a ver ahora como mostramos una u otra barra según la vista en la que nos encontremos. Para ello, cuando declaramos el *Pivot* en la página XAML capturamos el evento *LoadingPivotItem*:

```
<controls:Pivot Name="pivotViaje" LoadingPivotItem="cargandoPivotItem" FontSize="20">
```

Así, cada vez que se cambia de vista en el *Pivot*, se lanza este evento. Luego, en el *code behind* de la página, comprobaremos de qué vista se trata, para mostrar una barra de aplicación e incluso para no mostrar ninguna, como es en el caso de la última vista:

```

private void cargandoPivotItem(object sender, PivotItemEventArgs e)
{
    switch (((Pivot)sender).SelectedIndex)
    {
        case 0:
            AppBar = AppBarUno;
            break;

```

```

        case 1:
            ApplicationBar = AppBarDos;
            ApplicationBar.IsVisible = true;
            break;

        case 2:
            ApplicationBar.IsVisible = false;
            break;
    }
}

```

A cada vista del *Pivot* le corresponde un índice, así que preguntando por dicho índice, accederemos a cada uno de los casos del *switch*.

Antes de pasar a ver los comandos de la vista modelo que usamos en esta página, vamos a detenernos en el método *OnNavigatedTo()* del *code behind*. Este método los usamos, entre otras cosas, para recuperar la información guardada antes de salir de la aplicación. Esta información, para ser guardada, ha de ser serializada. El problema surge en esta vista porque para mostrar los gastos vamos a usar un nuevo elemento llamado *ExpanderView*. Este elemento lo hemos obtenido del paquete extra *Toolkit*, por lo que no es uno de los elementos que vienen por defecto en Silverlight. Esto implica que cuando queremos guardar la lista de elementos *ExpanderViews* que usamos para mostrar los gastos, no podemos hacerlo.

Para solucionar este problema, inmediatamente después de recuperar la información de la variable *State*, llamamos al método *mostrarGastos()*, que además de hacer otras comprobaciones referentes a la primera vista del *Pivot*, calcula la lista de gastos de nuevo. De esta forma conseguimos el mismo efecto que si hubiésemos serializado la lista:

```

contexto = (VMViajeConcreto)State["VMViajeConcreto"];
contexto.mostrarGastos();

```

b) Construcción de los elementos visuales de la vista lista de gastos: ExpanderView

Como acabamos de decir, en esta vista vamos a ver un elemento nuevo con el que rellenar el *ListBox*: se trata del elemento *ExpanderView*. Este elemento es otra de la novedades que podemos encontrar en el *Windows Phone Toolkit for Silverlight* del que ya hemos hablado anteriormente.

El *ExpanderView* muestra un encabezado y un contenido colapsable que se despliega al pulsar el encabezado. Se compone de dos partes: el encabezado o *header* que siempre está visible y los subelementos o ítems. A continuación se muestran dos imágenes. En la de la izquierda el *ExpanderView* está en su posición por defecto y en la de la derecha aparecen varios elementos expandidos (Ilustración 59).

Para consultar los gastos, el usuario simplemente tiene que seleccionar los encabezados que quiera y estos se desplegarán mostrando más información. Si quiere volver a ocultar la información solamente tiene que volver a pulsar en el encabezado.

Una vez introducido el nuevo elemento, vamos a ver el método *mostrarGastos()*. A este método lo llamaremos cada vez que se produzcan cambios que tengan que ver con los gastos, ya que en él calculamos la lista a mostrar:

```

public void mostrarGastos()
{
    using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
    {

```

```

if (DifGastoTotal < LimiteRojo)
{
    if (DifGastoTotal < 0)
    {
        Texto1 = "Ha sobrepasado ";
        Texto2 = " € el presupuesto máximo";
    }
    else
    {
        Texto1 = "Le quedan ";
        Texto2 = " € para llegar al máximo";
    }
    ColorGasto = "Red";
}
else
{
    ColorGasto = "White";
    Texto1 = "Le quedan ";
    Texto2 = " € para llegar al máximo";
}

```



Ilustración 60. Lista de gastos.

Primeramente abrimos la conexión a la base de datos y hacemos las comprobaciones necesarias para establecer el color de la cifra del gasto. Como estas condiciones las explicamos anteriormente (ver página 113) pasaremos directamente a ver la parte en la que calculamos la lista de gastos.

Primero accedemos a la base de datos para seleccionar una lista de todos los gastos correspondientes al viaje que estamos tratando. Después nos tenemos que crear una lista compuesta por elementos de tipo *ExpanderView*:

```

listaGastos = (from Gasto gasto in contextDb.Gastos where (gasto.ViajeId ==
parametro) select gasto).ToList();
List<ExpanderView> listaDEexpanderAUX = new List<ExpanderView>();

```

Una vez obtenida la lista con los gastos, la recorreremos. Y por cada gasto añadimos un nuevo encabezado a la lista de *ExpanderView* que nos creamos anteriormente:

```
for (int i = 0; i < listaGastos.Count; i++)
{
    ExpanderView expanderView = new ExpanderView();
    expanderView.Width = 500;
    expanderView.Header = listaGastos.ElementAt(i).Concepto;
```

Como vemos en el *for* del código, vamos a trabajar con un *ExpanderView* auxiliar llamado “aux”. En el encabezado asignamos el concepto del gasto.

A continuación nos creamos una lista de elementos “día” en la que guardar tantos días como tenga el viaje. Esta lista de días la obtenemos porque queremos mostrarle al usuario el número del día en el que se produjo el gasto, y en la tabla Gastos no se encuentra esa información:

```
listaDia = (from Dia dia in contextDb.Dias where (dia.DiaId ==
listaGastos.ElementAt(i).DiaId) select dia).ToList();
```

El siguiente paso es crear los subelementos, en los que tendremos dos *TextBox* debidamente rellenos con el día y el importe del gasto. Finalmente, añadir el elemento *ExpanderView* que nos acabamos de crear a la lista de *ExpanderViews*:

```
expanderView.Items.Add(new TextBlock() { Text = "Dia " +
listaDia.ElementAt(0).NumDia.ToString() });
expanderView.Items.Add(new TextBlock() { Text =
listaGastos.ElementAt(i).Importe.ToString() + " €" });
listaDExpander.Add(expanderView);
}
```

Una vez finalizado el bucle *for*, asignamos la lista de *ExpanderViews* auxiliar a la lista que realmente está enlazada con el elemento *ListBox* de la interfaz:

```
ListaDExpander = listaDExpanderAUX;
```

Una vez visto cómo se crean los elementos *ExpanderViews*, vamos a ver ahora cómo es el método que permite borrar estos gastos.

Para borrar un gasto de la lista, el usuario simplemente tendrá que seleccionarlo y pulsar el botón de eliminar que está integrado en la barra de aplicaciones. Una vez pulsado, si todo ha ido bien, se mostrará un mensaje indicando que el gasto ha sido borrado y, por supuesto, el gasto será eliminado de la lista.

El método que se encarga de eliminar el gasto es *EjecutarEliminarGasto()*. Una vez que empieza la ejecución de este método lo primero que tenemos que hacer es encontrar la forma de saber qué gasto es el que está seleccionado. En anteriores páginas esta tarea era sencilla ya que dentro de los controles *ListBox* existen propiedades como *SelectedItem* o *SelectedIndex* que nos permiten saber fácilmente cual es el elemento seleccionado. Pero en este caso hemos rellenado el *ListBox* con un control *ExpanderView* por lo que hay que buscar otra manera de saber el elemento seleccionado. La solución a este problema está en la consulta del *FocusManager*. La clase *FocusManager* proporciona útiles métodos relacionados con el foco de los elementos:

```
object focusObj = FocusManager.GetFocusedElement();
ListBoxItem itemSeleccionado = (focusObj as ListBoxItem);
```

En la variable “focusObj” guardamos el objeto que actualmente tiene el foco, es decir, el elemento que ha seleccionado el usuario. Posteriormente convertimos esa variable de tipo *object* en un objeto de tipo *ListBoxItem*, que es realmente lo que seleccionamos.

Como el ítem seleccionado es un objeto de tipo *ListBoxItem* tenemos que convertirlo en un objeto de tipo *ExpanderView* para que podamos acceder a sus subelementos y poder buscar así, con esos datos, en la base de datos el gasto a eliminar:

```
ExpanderView expanderViewAux = itemSeleccionado.DataContext as ExpanderView;
```

A continuación, obtenemos el concepto del gasto, accediendo directamente al encabezado del *ExpanderView*:

```
string gastoConcepto = expanderViewAux.Header.ToString();
```

Para conseguir extraer el número de día del gasto, nos crearemos un elemento de tipo *TextBlock* al que le asignaremos el primer subelemento del *ExpanderView*, a la vez que convertimos al objeto en uno de tipo *TextBlock*. A continuación, solamente tenemos que acceder a la propiedad *Text* del *TextBlock*, para tener una cadena en la que aparece el número de día. Finalmente, solo tenemos que eliminar las letras de la cadena para quedarnos con el número:

```
TextBlock textBlockDia = expanderViewAux.Items.ElementAt(0) as TextBlock;
String gastoDia = textBlockDia.Text;
gastoDia = gastoDia.Substring(4);
int numDiaSel = Convert.ToInt16(gastoDia)
```

La misma operación la repetimos para extraer el importe del gasto. Una vez obtenidos los datos, preguntamos al usuario si está seguro de eliminar el gasto. Para ello usamos un *MessageBox* con dos opciones de respuesta, una de confirmación y otra de cancelación:

```
MessageBoxResult message2 = MessageBox.Show("¿Esta seguro de querer eliminar el gasto " +
gastoConcepto + ", con importe " + cadenaImporte + " €?", "Eliminar gasto",
MessageBoxButton.OKCancel);
```

En caso de que la respuesta sea de cancelación, salimos del método inmediatamente. En caso de que sea de confirmación, continuamos ejecutando el código del método.

Como ya hemos obtenidos suficientes datos como para localizar el gasto inequívocamente en la base de datos, abrimos la conexión a la misma para obtener un día a partir del identificador de viaje que nos pasaron por parámetro en la navegación y el número de día que acabamos de obtener:

```
using (AppDbContext contextDb = new AppDbContext("Data
Source='isostore:/MiViajeDb.sdf'"))
{
    listaDiaAux = (from Dia dia in contextDb.Dias where (dia.ViajeId ==
parametro && dia.NumDia == numDiaSel) select dia).ToList();
```

A partir del día obtenido accedemos a su campo "IdDia" para buscar el gasto que queremos borrar junto con el identificador de viaje, el importe y el concepto:

```
List<Gasto> gastoBorrar = (from Gasto gasto in contextDb.Gastos
where
    (gasto.ViajeId == parametro &&
gasto.DiaId == listaDiaAux.ElementAt(0).DiaId &&
gasto.Importe == importeSel &&
gasto.Concepto == gastoConcepto)
select gasto).ToList();
```

De esta manera hemos obtenido un solo gasto, el cual ya solo nos queda ponerlo en estado de pendiente eliminación y notificar los cambios a la base de datos:

```
contextDb.Gastos.DeleteAllOnSubmit(gastoBorrar);
contextDb.SubmitChanges();
```


Como el gasto ya ha sido eliminado, la cifra que nos indica la cantidad de dinero que nos queda hasta llegar al máximo ha sido modificada. Esta vez, esa cifra aumentará, ya que al gasto total, se le restará el importe del gasto eliminado:

```
gastoTotal = gastoTotal - importeSel;  
DifGastoTotal = Presupuesto - gastoTotal;  
ValorAbsolutoMostrado = Math.Abs(DifGastoTotal);
```

Por último volvemos a llamar al método *mostrarGastos()*, que se encarga de pintar la cifra de la que hablábamos anteriormente en banco o en rojo según corresponda y de actualizar la lista de gastos, que esta vez, contendrá un gasto menos.

c) Brújula

Como mencionamos anteriormente, esta parte de la aplicación se pretende que sea usada mientras el usuario está realizando el viaje, y una de las mejores ayudas que podemos brindar cuando se está en un sitio desconocido, es en la localización. Para eso vamos a incluir en la tercera vista del *Pivot* un lanzador que nos muestra rutas con indicaciones entre dos puntos. Además incluiremos una brújula digital que simula las brújulas magnéticas (Ilustración 60):



Ilustración 61. Vista navegación con la brújula habilitada.

Como se ve en la anterior imagen, el usuario va a disponer de dos elementos de tipo *TextBox* para introducir un punto de partida y otro de destino. Una vez rellenados ambos campos pulsará el botón “Ver ruta”.

A la derecha del botón “Ver ruta” vamos a mostrar la información relativa a la desviación en grados respecto al norte que nos ofrece la brújula. Ésta se mostrará en la parte inferior de la pantalla una vez que el usuario haya calibrado el dispositivo.

A la hora de usar la brújula debemos tener en cuenta que, a diferencia de otros sensores como el acelerómetro, este sensor no está incluido en los requisitos mínimos de Windows Phone (ver página 9). Aunque es muy corriente encontrarlo, ocasionalmente puede haber dispositivos que no lo incorporen. Por esta razón vamos a comprobar su disponibilidad antes de usarla.

También tenemos que tener en cuenta que la API de la brújula solo funciona correctamente cuando el dispositivo está en posición horizontal.

Lo primero que debemos hacer para usar la API es añadir una referencia en nuestro proyecto al ensamblado *Microsoft.Devide.Sensors*. A continuación vemos el código XAML encargado de dibujar la brújula:

```
<Grid Grid.Row="1" >
    <Ellipse Fill="Black" Margin="83,0,83,0" ></Ellipse>
    <Ellipse Margin="88,5,88,5">
        <Ellipse.Fill>
            <ImageBrush
                ImageSource="/AgendaViajesUc3m;component/Imagenes/Compass_T_E.png" />
        </Ellipse.Fill>
    </Ellipse>
    <Grid.Projection>
        <PlaneProjection x:Name="GridProjection">
        </PlaneProjection>
    </Grid.Projection>
</Grid>
```

En el código anterior se ve que nos hemos creados dos elementos de tipo *Ellipse*. La primera de ellas, más grande, la usaremos como borde, mientras que la segunda incluirá la imagen de una brújula. Dentro de las elipses, la característica más importante es la propiedad *Image.Projection*. En ella creamos un nuevo elemento *PlaneProjection*, el cual nos ofrece propiedades para rotar y trasladar el objeto al que se lo apliquemos.

Toda la lógica del manejo de la brújula la vamos a realizar en el *code behind*. Empezaremos viendo el constructor, en el que comprobaremos si el dispositivo dispone del sensor necesario (brújula):

```
if (Compass.IsSupported)
{
    brujula = new Compass();
    brujula.TimeBetweenUpdates = TimeSpan.FromMilliseconds(1);
    brujula.CurrentValueChanged +=
        new EventHandler<SensorReadingEventArgs<CompassReading>>
            (brujula_CurrentValueChanged);

    brujula.Calibrate +=
        new EventHandler<CalibrationEventArgs>(brujula_Calibrate);
    brujula.Start();
}
else
{
    MessageBox.Show(@"Su dispositivo no tiene brújula integrada
                    o no está disponible.");
}
```

Después de crear la nueva instancia de la brújula, lo primero que hacemos es establecer la propiedad *TimeBetweenUpdates*, que indica el tiempo que pasará entre actualizaciones de las lecturas. También estableceremos el evento *CurrentValueChanged* que se lanzará cada vez que se actualicen los valores recogidos por el sensor. Por último, llamamos al método *Start* que inicia la brújula. Vemos cómo, en caso de que el sensor no esté disponible, mostraríamos un mensaje indicándolo.

Algo muy importante a tener en cuenta, es que el manejador de eventos *brujula_CurrentValueChanged* se ejecutará en un hilo diferente al de la aplicación por la forma de funcionar del sensor de la brújula. Por lo

tanto, para poder actualizar la interfaz de usuario, deberemos invocar a un método que se ejecute en el hilo de aplicación usando el objeto *Dispatcher*:

```
void brujula_CurrentValueChanged(object sender,
SensorReadingEventArgs<CompassReading> e)
{
    Dispatcher.BeginInvoke(() => UpdateUI(e.SensorReading));
}
```

Se trata del método *UpdateUI*, al que le pasamos la lectura recibida del sensor. Es este método el encargado de procesar esa lectura y usarla tanto para girar la brújula, como para mostrar los grados de inclinación respecto al norte. Esta última información la mostramos al usuario en pantalla a través de un *TextBlock* llamado “grados”. Pero antes, tenemos que comprobar si estamos en medio de la calibración, en cuyo caso, comprobamos si estamos en un valor de precisión por debajo de 15 grados. Si se cumple esta condición, ocultamos el aviso (Ilustración 61) y permitimos recibir valores al resto de nuestro código:



Ilustración 61. Aviso para recalibrar la brújula.

```
private void UpdateUI(CompassReading reading)
{
    if (rectCalibrating.Visibility == System.Windows.Visibility.Visible)
    {
        if (reading.HeadingAccuracy < 15)
            ShowCalibration(false);
        else
        {
            GridProjection.RotationZ = reading.MagneticHeading;
            grados.Text = Math.Round(reading.MagneticHeading, 2).ToString();
        }
    }
}
```

Como hemos visto en el código anterior, tenemos que tener en cuenta que es posible que tengamos que calibrar la brújula para su correcto funcionamiento.

Cuando sea necesario se lanzará el evento *Calibrate*:

```
brujula.Calibrate += new EventHandler<CalibrationEventArgs>(brujula_Calibrate);
```

Debemos manejar este evento y, cuando nos llegue, mostrar al usuario instrucciones para poder calibrar su dispositivo (Ilustración 62).

No recibiremos ninguna confirmación de que la calibración ha terminado, somos nosotros los encargados de determinar cuándo hemos obtenido la precisión necesaria para trabajar con el sensor, mediante la propiedad *HeadingAccuracy* de la clase *CompassReading*.

Por norma general, en aplicaciones que no necesiten ser extremadamente precisas, podemos decir que una precisión de entre 10 y 15 grados es suficiente para terminar la calibración.



Ilustración 62. Pedimos al usuario que calibre el sensor realizando un movimiento.

Al igual que en el manejador *brujula_CurrentValueChanged*, *brujula_Calibrate* se ejecuta en un hilo separado del principal de la aplicación, por lo que debemos usar el objeto *Dispatcher* para invocar a un método que se ocupe de mostrar el aviso al usuario:

```
void brujula_Calibrate(object sender, CalibrationEventArgs e)
{
    Dispatcher.BeginInvoke(() => ShowCalibration(true));
}
```

En nuestro caso, el método *ShowCalibration* nos servirá para mostrar u ocultar el aviso en pantalla de que necesitamos la calibración:

```
private void ShowCalibration(bool show)
{
    if (show)
        rectCalibrating.Visibility = System.Windows.Visibility.Visible;
    else
        rectCalibrating.Visibility = System.Windows.Visibility.Collapsed;
}
```

d) Vista modelo de la tercera vista del Pivot: navegación

Para realizar el cálculo de rutas vamos a usar la misma técnica de enlace a datos que usamos en el resto de la aplicación, por lo que veremos en la vista modelo cómo calcular las rutas. En este apartado explicaremos cómo usar un lanzador que nos permita acceder al sistema de cálculo de rutas que nos ofrece el sistema operativo.

Un lanzador es una llamada a una API de Silverlight que ejecuta una acción en el sistema operativo de la cual no necesitamos ni esperamos ninguna respuesta.

Al ejecutar un lanzador salimos de nuestra aplicación, por lo que entra en funcionamiento el ciclo *tombsoning*. Un ejemplo de lanzador es cuando indicamos al sistema que abra el navegador de internet y

cargue una página web. De esta acción no recibiremos ninguna respuesta y puede que el usuario nunca vuelva a nuestra aplicación. Al contrario que los selectores, en los que tras interoperar con el sistema operativo, regresaremos a la aplicación, como era el caso de la cámara de fotos.

Al igual que en el resto de páginas de la aplicación, hemos usamos la técnica de enlace a datos para pasar a la vista modelo directamente los datos que introduzca el usuario. Por otro lado, el botón “Ver ruta”, lo vincularemos a un comando delegado:

```
public DelegateCommand CalcularRuta
{
    get
    {
        if (calcularRuta == null)
            calcularRuta = new DelegateCommand(EjecutarCalcularRuta,
                                                PuedeEjecutarCalcularRuta);

        return calcularRuta;
    }
}
```

El método *PuedeEjecutarCalcularRuta()* devuelve un booleano con valor *true* siempre que ni el punto inicial ni el punto final estén vacíos:

```
public bool PuedeEjecutarCalcularRuta()
{
    bool devuelto = true;
    if (ptoInicial == String.Empty || PtoFinal == String.Empty)
    {
        devuelto = false;
    }
    return devuelto;
}
```

Dentro del método *EjecutarCalcularRuta()* vamos a llamar dos veces al método *calcularCoordenadas(String punto)*, en el cual creamos peticiones para consultar con el *Bing Maps SOAP Services* las coordenadas de los puntos que le pasamos por parámetro. Primero lo invocamos pasándole la información del punto de partida y, una vez hecha la consulta, volvemos a llamar al método, esta vez pasándole la información del punto de destino:

```
public void EjecutarCalcularRuta()
{
    calcularCoordenadas(ptoInicial);
    if (inicialCompletado)
    {
        calcularCoordenadas(ptoFinal);
    }
}
```

El método *calcularCoordenadas()* es exactamente el mismo que se usa en la vista modelo de la página *EditaPlan.xaml* (consultar página 78). Para no repetir código, podríamos haber incluido el método en la clase “VMBase” ya que ésta es una clase implementada por todas las vistas modelos de la aplicación. Pero dentro de este método llamamos al método *geocodeService_GeocodeCompleted()*, y este método sí que es distinto por el siguiente motivo.

Existe un problema cuando trabajamos con los *SOAP Services* que nos ofrece Microsoft: su código tarda más en ejecutarse que el propio código de la aplicación. La información que tiene que consultar es remota, por lo que si hubiésemos llamado dos veces seguidas al método *calcularCoordenadas()*, no daríamos tiempo a que recuperase la información que le pedimos en la primera llamada, cuando ya estamos solicitando la información de la segunda. Por este motivo, el método *geocodeService_GeocodeCompleted()* es distinto en la vista modelo “VMEeditaPlan”. En

“VMViajeConcreto” le hemos hecho unas modificaciones para que no se ejecute hasta estar completamente seguros de que hemos recibido correctamente la información de la primera llamada:

```
void geocodeService_GeocodeCompleted(object sender, GeocodeCompletedEventArgs e)
{
    if(!inicialCompletado){
        ObtenerResultado(coordenadasOrigen,e);
        inicialCompletado = true;
    }

    if (inicialCompletado)
    {
        ObtenerResultado(coordenadasDestino,e);
        BingMapsDirectionsTask ruta = new BingMapsDirectionsTask();
        ruta.Start = new LabeledMapLocation(ptoInicial, coordenadasOrigen);
        ruta.End = new LabeledMapLocation(ptoFinal, coordenadasDestino);
        ruta.Show();
    }
}
```

Como se ve en el código, usamos una variable booleana para avisarnos cuando se haya ejecutado la primera consulta, y en el momento que termina, volvemos a llamar al método *ObtenerResultados*(*GeoCoordinate* coordenadas, *GeocodeCompletedEventArgs* e), esta vez pasándole las coordenadas del punto de destino. Ahora sí, el método *ObtenerResultados*(*GeoCoordinate* coordenadas, *GeocodeCompletedEventArgs* e) es el mismo que usamos en la vista modelo “VMEditanPlan” (consultar página 99).

Una vez obtenidos los resultados de ambas consultas, creamos una nueva instancia del lanzador *BingMapsDirectionsTask*, establecemos los puntos de inicio y fin y por último llamamos al método *Show*, que mostrará la aplicación de *Bing Maps* con la ruta que hemos indicado.

Es muy importante recordar que, al ejecutar un lanzador, nuestra aplicación se cerrará entrando en estado durmiente y existiendo la posibilidad de que se realice el *tombstoning*, por lo que debemos guardar todos los datos necesarios para restablecerla si fuese necesario.

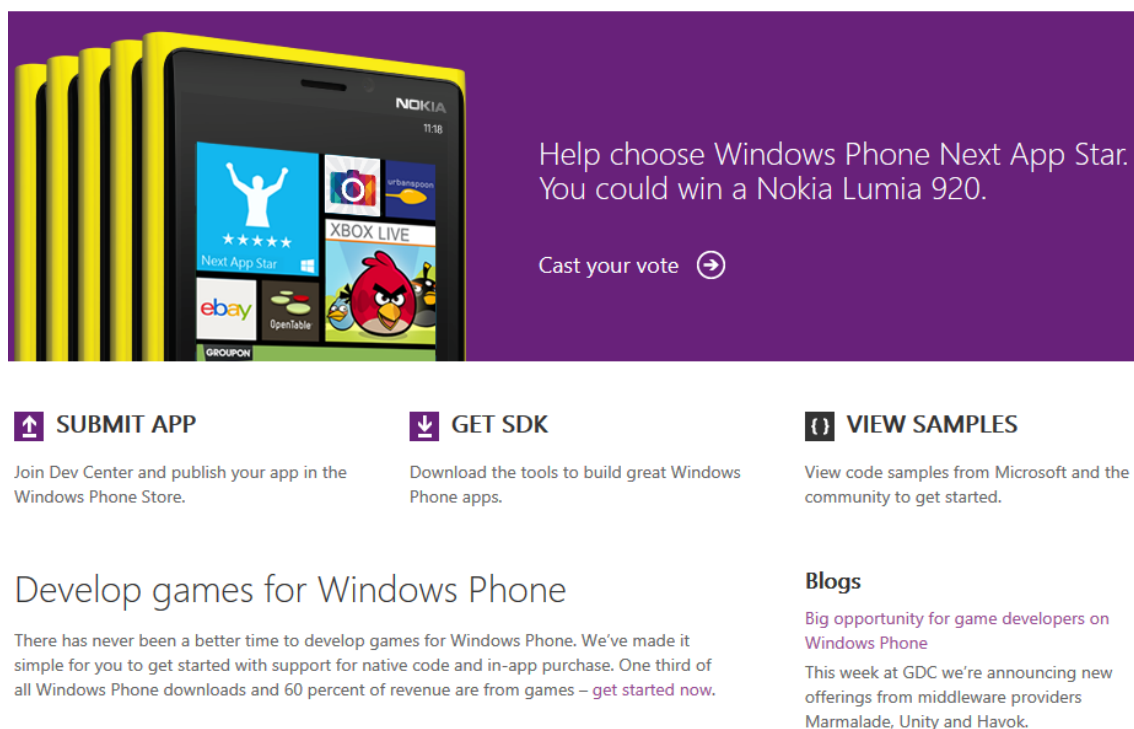
5. PUBLICACIÓN DE UNA APLICACIÓN EN EL MARKETPLACE

Una vez acabada la aplicación, el siguiente paso es publicarla en el Marketplace de Windows Phone para que cualquier usuario pueda descargarla.

Este proceso incluye tres pasos. En primer lugar, hay que registrarse y obtener una cuenta autorizada del Marketplace. A continuación hay que probar la aplicación para asegurarnos de que cumple las políticas de certificación de Microsoft. Por último, debemos publicar la aplicación.

I. Crear una cuenta de Marketplace

Para poder publicar aplicaciones en el Marketplace de Windows Phone, debemos crear una cuenta en el Dev Center, el centro desde el cual se puede gestionar nuestras aplicaciones, obtener estadísticas y gestionar los pagos y cobros. Para ello, vamos a su página web [4] (Ilustración 63):



Help choose Windows Phone Next App Star. You could win a Nokia Lumia 920.

Cast your vote →

↑ SUBMIT APP
Join Dev Center and publish your app in the Windows Phone Store.

↓ GET SDK
Download the tools to build great Windows Phone apps.

VIEW SAMPLES
View code samples from Microsoft and the community to get started.

Develop games for Windows Phone
There has never been a better time to develop games for Windows Phone. We've made it simple for you to get started with support for native code and in-app purchase. One third of all Windows Phone downloads and 60 percent of revenue are from games – [get started now](#).

Blogs
[Big opportunity for game developers on Windows Phone](#)
This week at GDC we're announcing new offerings from middleware providers Marmalade, Unity and Havok.

Ilustración 63. Centro de desarrollo de Windows Phone.

Pulsando sobre el enlace “SUBMIT APP”, llegaremos hasta la página de entrada del registro.

Para comenzar el registro, debemos pulsar sobre el enlace “Join Now” que hay en la parte derecha de la página. Esto nos llevará directamente al proceso de registro, compuesto de cinco pasos: Tipo de cuenta, detalles, perfil, pago y confirmación.

Una vez registrado, obtenemos acceso a publicar aplicaciones o juegos para Windows Phone y para Xbox360, todo en la misma cuenta.

a) Tipo de cuenta

En este paso del registro tendremos que seleccionar el país de residencia y seleccionar el tipo de cuenta que deseamos registrar. Podemos elegir entre compañía o individual / estudiante.

Una vez que hayamos escogido el tipo de cuenta, solo tenemos que aceptar las condiciones y términos de uso del Dev Center y marcar la casilla de “Legal Terms” que tenemos en la parte inferior del formulario. Ahora presionamos el botón “Next” para ir al siguiente paso: los detalles de la cuenta.

b) Detalles de la cuenta

Este paso del registro también es muy sencillo, solo nos piden los datos personales: nombre, apellidos, dirección, teléfono, etc. Más adelante se verificarán estos datos y si no coinciden o no son reales, se podría cancelar la cuenta del Marketplace.

En este paso también debemos indicar el “Publisher name”, es decir, el nombre bajo el cual se publicarán nuestras aplicaciones en el Marketplace y que todo el mundo podrá ver.

Una vez rellenados estos datos, pasamos a la siguiente pantalla pulsando de nuevo el botón “Next”.

c) Opciones de pago

En el último paso deberemos confirmar si deseamos pagar los 75 €, si somos estudiantes o si deseamos canjear un cupón descuento. En nuestro caso, elegimos la opción “I’m student” y presionamos el botón “Next” para finalizar la inscripción.

II. Probando la aplicación

Una vez obtenida la cuenta en el App Hub y terminado de desarrollar la aplicación, es momento de comprobar que todo funciona como debe.

Lo primero que tenemos que hacer es compilar el proyecto para generar el fichero en formato XAP. Esto lo conseguimos cambiando la opción del combo que está en la barra de herramientas a la derecha del selector de destino (Ilustración 64)

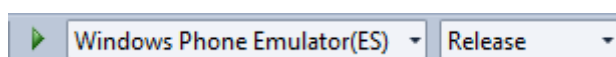


Ilustración 64. Selector de solución.

Tras seleccionar la opción “Release” en el combo pulsaremos la tecla F6 de nuestro teclado para compilar el proyecto. Esta acción generará nuestro archivo XAP en la ruta D:\NombreDelProyecto\Bin\Release\.

Para publicar una aplicación en el Marketplace de Windows Phone ésta debe pasar un proceso de certificación. En este proceso se realizan pruebas de funcionamiento a la aplicación y se examina el código para asegurar que cumple con los requisitos de calidad de Microsoft.

Como ya se indicó al principio, junto con las herramientas de desarrollo de Windows Phone 7.5, Microsoft ha incluido el “Marketplace Test Tool”, una serie de pruebas automáticas y manuales que imitan las realizadas durante el proceso de certificación de las aplicaciones.

Para acceder a esta herramienta, tendremos que abrir nuestro proyecto en Visual Studio 2010, presionar con el botón derecho del ratón sobre el proyecto en el explorador de soluciones y escoger la opción “Abrir el Kit de pruebas de Marketplace” (Ilustración 65):

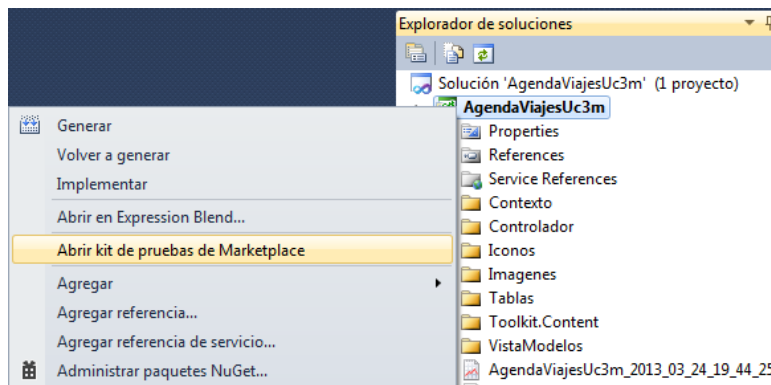


Ilustración 65. Abrir kit de pruebas de Marketplace.

a) Detalles de la aplicación

Lo primero que nos pedirá es que rellenemos los detalles de la aplicación, de la misma forma que haríamos en el Marketplace: Indicar el .XAP del proyecto, los iconos y al menos una captura de pantalla de la aplicación (Ilustración 66):



Ilustración 66. Detalle de la aplicación para iniciar el kit de pruebas.

La imagen (Tile) grande de aplicación es la que se mostrará cuando la aplicación esté anclada a la pantalla de inicio. Debe tener un tamaño de 173x173 píxeles. La siguiente es la imagen pequeña de la aplicación. Es la que se usará en la lista de aplicaciones del dispositivo, debe tener un tamaño de 62x62 píxeles. Por último, la imagen para el Marketplace, es la que se mostrará en la tienda de aplicaciones, en Zune o en la web y debe tener un tamaño de 200x200 píxeles.

Las capturas de pantalla deben tener un tamaño exacto de 480x800 píxeles y solo deben incluir la pantalla de la aplicación, no el marco del emulador. Es muy importante que no se modifiquen de ninguna forma. Si realizamos algún retoque, aunque simplemente sea añadir un texto sobre estas pantallas, nuestra aplicación será rechazada.

b) Pruebas automáticas

El siguiente paso de la herramienta son las pruebas automatizadas. Verifican varios aspectos de la información ofrecida en el paso anterior (Ilustración 67):

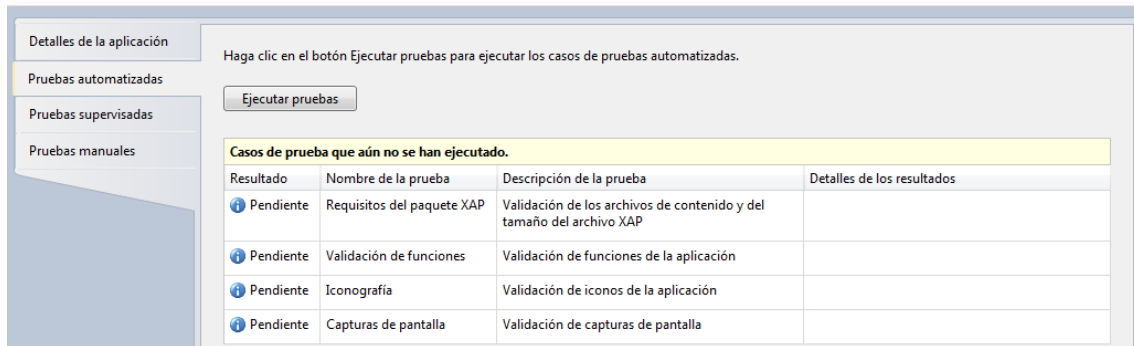


Ilustración 67. Inicio de las pruebas automatizadas.

Estas pruebas comprueban que el tamaño y contenidos del paquete XAP sea correctos, que las capacidades necesarias para ejecutar la aplicación estén correctamente indicadas y la validez de iconos y de las capturas de pantalla (Ilustración 68):

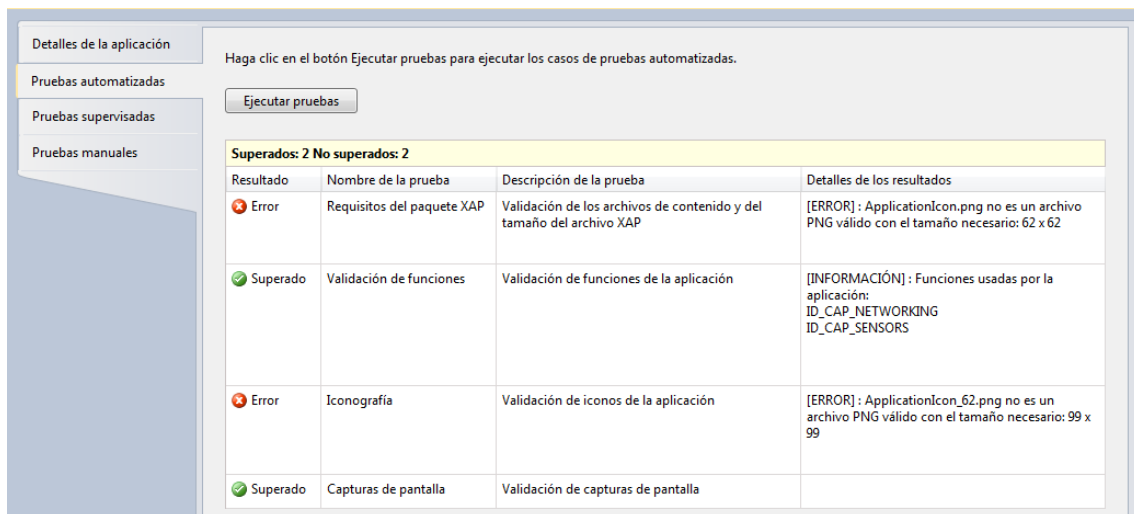


Ilustración 69. Centro de desarrollo de Windows Phone.

Si alguna prueba falla, en la columna de detalles “Result Details” nos dará información sobre qué ha fallado.

c) Pruebas supervisadas

Este paso de las pruebas requiere que tengamos un dispositivo registrado para desarrollo y conectarlo al PC (Ilustración 70).

Este proceso desplegará la aplicación en el dispositivo, que tendrá que estar conectado al PC mediante un cable USB y con la pantalla de inicio del teléfono desbloqueada. Debemos usar la aplicación de forma normal, navegando entre páginas, accediendo a todas las características, incluso implicando el ciclo de vida, saliendo al inicio y volviendo a la aplicación o usando las características de red o lanzadores y selectores que tenga nuestra aplicación. Cuando se salga de la aplicación, usando el botón atrás, se refrescarán los resultados, indicando si la aplicación consume demasiada memoria, si el tiempo de lanzamiento supera el permitido y si el uso del botón atrás es correcto (Ilustración 71).

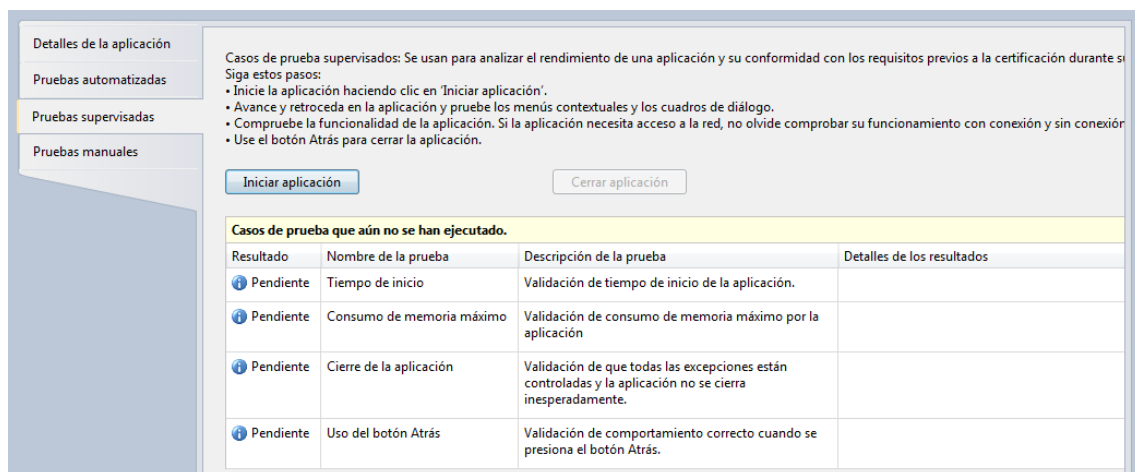


Ilustración 70. Inicio de las pruebas supervisadas.

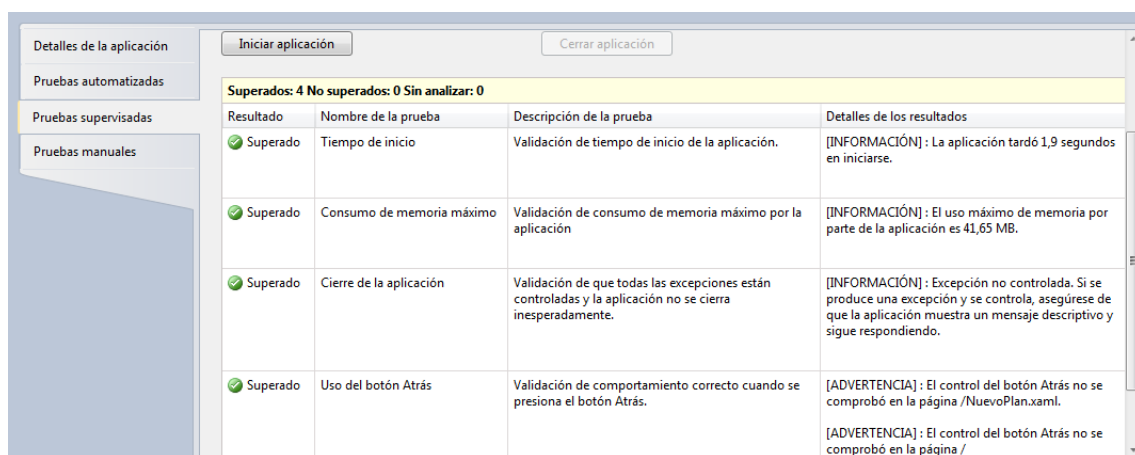


Ilustración 71. Resultado de las pruebas supervisadas.

Si alguna de estas pruebas fallara, nunca podremos publicar la aplicación en el Marketplace por lo que debemos cumplir estas todas estas especificaciones obligatoriamente.

d) Pruebas manuales

Esta última parte del kit de pruebas, las pruebas manuales, nos propone 50 casos de prueba que debemos verificar. En cada prueba encontraremos una columna para indicar el resultado, una columna con el nombre de la prueba y por último, una columna con las instrucciones que debemos seguir para realizar la misma (Ilustración 72).

El objetivo es que cuantas más pruebas pasemos satisfactoriamente, más seguridad tendremos que la aplicación sea aprobada en el primer intento.

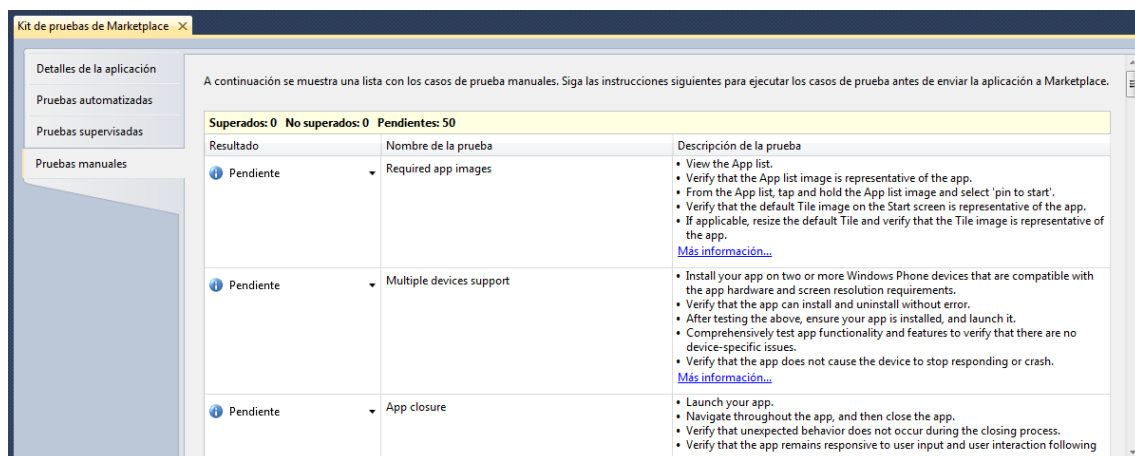


Ilustración 72. Inicio de las pruebas manuales.

III. Publicando la aplicación

Al terminar de pasar las pruebas de manera satisfactoria, nos dirigimos de nuevo a la página del Dev Center [4] . Nos identificaremos con la cuenta de Microsoft que usamos anteriormente para registrarnos en él.

Una vez autenticados, pulsamos el enlace “SUBMIT APP” que nos dirigirá al *dashboard* de Windows Phone (Ilustración 73):

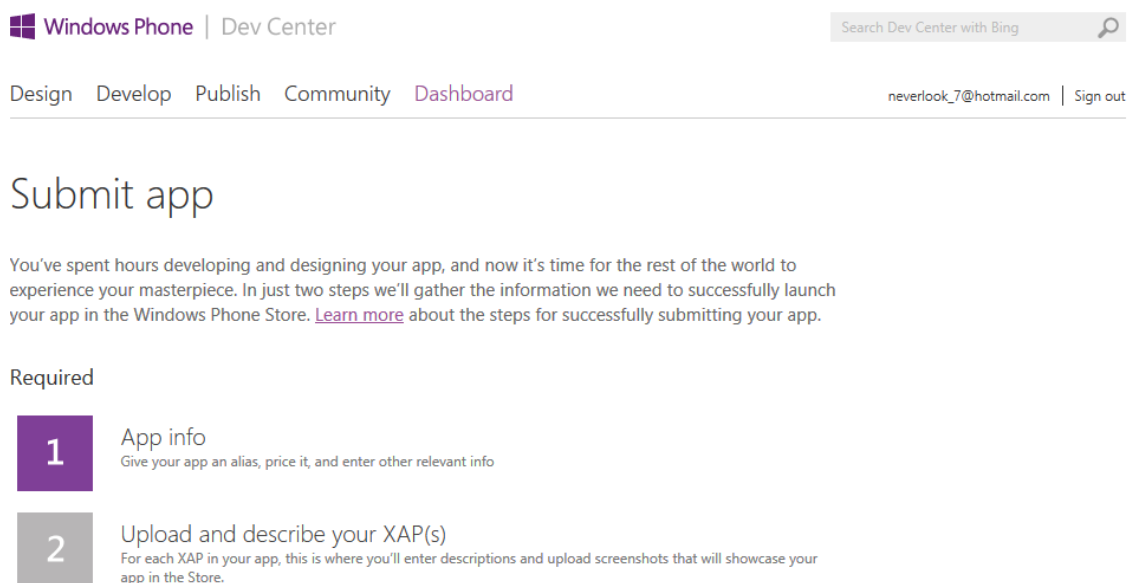


Ilustración 73. Dashboard de Windows Phone.

El primer paso consiste en indicar el nombre de la aplicación, la categoría y el precio. Antes de guardar estos datos, marcamos la opción “Distribute to all markets except China” para que nuestra aplicación esté disponible en el mercado internacional.

En el segundo paso se nos pedirá subir el archivo XAP correspondiente a la aplicación, indicar la versión de la misma y establecer una descripción para que el usuario se haga una idea de lo que ofrece la aplicación. Además se nos pedirá indicar las palabras clave que ayudarán a los usuarios a encontrar nuestra aplicación en el Marketplace.

Por último, deberemos cargar tres tipos de imágenes:

- La imagen del “Tile Icon”, es decir el icono de nuestra aplicación, que deberá ser de 300x300 píxeles.
- La imagen de fondo. Esta imagen será de 1000x800 píxeles y aparecerá de fondo en la página de nuestra aplicación en Zune.
- Las diferentes capturas de pantalla de nuestra aplicación. Estas imágenes se le mostrarán al usuario para que observe una vista previa de la aplicación antes de descargársela.

Una vez terminado el segundo paso, seleccionamos el enlace opcional “Map services” ya que en nuestra aplicación hacemos uso de mapas (Ilustración 74):

Optional



Add in-app advertising
Getting paid through ads? It's all here.



Market selection and custom pricing
For apps, you have the option to define different pricing and availability for different countries/regions.




Map services
Get the token required to use map services in your app.

Ilustración 74. Opciones de aplicación.

Habrà que indicar la clave de Bings Maps que anteriormente conseguimos para poder utilizar los mapas de Bings.

Una vez introducidos los credenciales presionamos el botón “Save” y tendremos subida completamente nuestra aplicación (Ilustración 75):

 Windows Phone | Dev Center

Search Dev Center with Bing 

Design Develop Publish Community **Dashboard**

neverlook_7@hotmail.com | Sign out

App submission successful!

Agenda de viajes Uc3m

If your app needs to be tested, it will take us up to 5 business days to make sure it meets our certification requirements. If not, within 24 hours you can either publish the change yourself or it will go live automatically. We'll send you an email when your app is ready or if we have more questions about your submission.

Would you like to...

[Submit another app](#)

[Go to the Lifecycle page](#)

[Go to the Dashboard](#)

Ilustración 75. Aplicación subida correctamente.

Una vez hecho esto, comenzará el proceso de certificación. Solo tenemos que esperar, y si todo está correctamente, la aplicación estará disponible en el Marketplace en 3 o 5 días laborables.

Tras este breve periodo de tiempo recibiremos un correo electrónico en la dirección con la que nos hayamos registrado indicándonos que la aplicación ha pasado satisfactoriamente el proceso de certificación y estará disponible en el Marketplace.

Así, nuestra aplicación está disponible como cualquier otra aplicación del Marketplace con su propia URL [9].

6. CONCLUSIONES

El principal objetivo de este proyecto era enseñar a crear aplicaciones para quien desee empezar a desarrollar en Windows Phone, de modo que tras ver todo el proyecto podemos considerar que el objetivo se ha cumplido. Hemos creado una aplicación con variedad de funcionalidades explicando cada paso y cada elemento nuevo que utilizábamos, de forma que puedan extrapolarse a otras aplicaciones.

También hemos dedicado unos capítulos a explicar cómo el sistema operativo gestiona sus recursos para interactuar con la aplicación de forma que podamos sacarle el mayor partido posible. De ese modo hemos podido aplicar ciertos métodos para conseguir optimizar el código todo lo posible, teniendo siempre en cuenta este tema ya que es primordial cuando se trabaja en dispositivos móviles, cuyos recursos son más limitados que los de ordenadores de sobremesa.

Al profundizar en el desarrollo de aplicaciones para este sistema operativo hemos descubierto multitud de métodos y APIs singulares que permiten implementar funcionalidades de todo tipo. En este proyecto hemos explicado algunas de ellas, como es trabajar con bases de datos, añadir funcionalidades de GPS u obtener información de servicios web. Pero hay muchas más APIs interesantes que poder estudiar como programar tareas en el teléfono, interactuar con el acelerómetro, consumir RSS o extraer información de archivos XML. Para ello se pueden consultar varios de los libros que están indicados en la sección bibliográfica

Por último hemos explicado cómo publicar una aplicación en el Marketplace, que es el objetivo de todo desarrollador: el de poder exponer su trabajo para que pueda ser usado y valorado por los usuarios de este sistema operativo.

7.PRESUPUESTO

Ya que el objetivo de este proyecto es el de crear una aplicación Software, los principales gastos para llevarlo a cabo han sido de licencias y personal.

- Duración: 14 meses.
- Presupuesto total: 21293,625 €
- Desglose presupuestario:

PERSONAL

Apellidos y Nombre	N.I.F.	Dedicación	Coste hombre mes
Castellanos de la Torre, Fco. José	00000000X	14 meses	1.500 €
			Total: 21.000 €

EQUIPOS

Descripción	Coste (Euros)	% Uso dedicado proyecto	Dedicación (meses)	Período de depreciación (meses)	Coste imputable ³
Notebook HP DV63181	658,48	100	14	60	153,64 €
Nokia Lumia 800	394,21	100	8	36	87,6 €
					Total: 241,24 €

OTROS COSTES

Descripción	Coste (Euros)	% Uso dedicado proyecto	Dedicación (meses)	Período de depreciación (meses)	Coste imputable
Microsoft Office 2010	97,58	100	14	60	22,76 €
Registro en Windows Phone Dev Center	59,25	100	6	12	29,625 €
					Total: 52,385 €

³ Fórmula del cálculo de la amortización: $\frac{A}{B} \times C \times D$

A: Número de meses desde la fecha de facturación en que el equipo es utilizado.

B: Período de depreciación.

C: Coste del equipo, sin IVA.

D: Porcentaje del uso que se dedica al proyecto.

8.REFERENCIAS

a) Sitios web

- [1] Nuget Gallery, «Install Package,» [En línea]. Available: <http://www.nuget.org>.
- [2] J. Grossman, «Introduction to Model/View/ViewModel pattern for building WPF apps,» 08 Octubre 2005. [En línea]. Available: <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>.
- [3] M. Fowler, «Blog de Martin Fowler,» 2004. [En línea]. Available: <http://martinfowler.com/eaaDev/PresentationModel.html>.
- [4] Windows Phone, «Dev Center,» [En línea]. Available: <https://dev.windowsphone.com/en-us>.
- [5] Code Plex, «Project Hosting for Open Source Software,» [En línea]. Available: <http://appbarutils.codeplex.com>.
- [6] Microsoft, «Msdn,» [En línea]. Available: <http://msdn.microsoft.com/en-us/library/cc980922.aspx>.
- [7] Microsoft, «Msdn Library,» [En línea]. Available: <http://msdn.microsoft.com/en-us/library/cc981067.aspx>.
- [8] Bing, «Maps Account Center,» [En línea]. Available: <http://www.bingmapsportal.com/>.
- [9] J. Ferguson, B. Petterson y J. Beres, La biblia de C#, Anaya.
- [10] C. Petzold, «Data bindings,» de *Programming Windows Phone 7*, Redmon, Washington, Microsoft Press, 2010, p. 1007.

b) Libros consultados

Shawn Wildermuth: "Windows Phone 7.5. Desarrollo de aplicaciones con Silverlight". Ed.: Anaya

Daniel Vaughan: "Windows Phone 7.5 Unleashed". Ed.: Sams

Josué Yeray e Ibon Landa: "Windows Phone 7.5. Desarrollo de aplicaciones en Silverlight". Ed.: Krasis Press.

Charles Petzold: "Programming Windows Phone 7". Ed.: Microsoft Press

J. Ferguson, B. Petterson y J. Beres: "La biblia de C#". Ed.: Anaya